M6800 Family Assembler

User's Manual

David C. Pheanis
Western Microsystems

Third Edition
January, 2003

I have used reasonable care to make the assembler and this manual as accurate and as usable as possible, but I have probably overlooked a few mistakes. In the interest of quality, therefore, I am offering a reward of $1.00 in U.S. funds to the first finder of each error, whether it is technical, typographical, grammatical, or otherwise. Any student who uses the assembler and this manual in my class may elect to receive 20 points of class credit in lieu of a cash reward for reporting an error. I hope that this offer will encourage people to report my mistakes so I can correct them.

I shall appreciate receiving positive suggestions for improving this manual or the assembler in any way. Many of the features that are already implemented in the assembler evolved from discussions with users, and future suggestions for improvements will be more than welcome.

Disclaimer: The reward that I mentioned above applies only to the first four chapters. The rest of the manual is in preliminary form, and I added it rather hastily in an effort to make the manual as complete as possible as soon as possible.

— DCP

Copyright © 1985, 2003 by David C. Pheanis.

All rights reserved. No part of this publication may be reproduced, stored in any retrieval system, photocopied, recorded, or otherwise duplicated or transmitted in any form or by any means, electronic, mechanical, or otherwise, without the prior written permission of the author.

Contents

1 Introduction **1**

2 Source Syntax **5**

2.1 Source Characters 6

2.2 Comments 6

2.3 Label Field 8

2.3.1 Symbolic Labels 8

2.3.2 Local Labels 9

2.3.3 Conditional Labels 12

2.4 Opcode Field 12

2.5 Operand Field 13

2.6 Comment Field 13

2.7 Statement Continuation 14

2.8 Assembler Support for Text Formatters 17

2.8.1 Dot Commands 18

2.8.2 Grave Accents 19

3 Addressing Modes for Executable Instructions **21**

3.1 Inherent Addressing 21

3.2 Immediate Addressing 21

3.3 Indexed Addressing 22

3.4 Extended Addressing 23

3.5 Direct Addressing 23

3.6 Self-Relative Addressing 24

4 Expressions **27**

4.1 Terms . . . 27

4.1.1 Binary Numbers . . . 27

4.1.2 Octal Numbers . . . 28

4.1.3 Decimal Numbers . . . 28

4.1.4 Hexadecimal Numbers . . . 28

4.1.5 Symbolic Labels . . . 29

4.1.6 Local Labels . . . 29

4.1.7 Quoted ASCII Characters . . . 30

4.1.8 Present-Location Symbol . . . 30

4.1.9 RAD-40 Hash String . . . 31

4.1.10 Conditional-Assembly Mask . . . 32

4.1.11 Parenthesized Subexpression . . . 33

4.2 Operators . . . 33

4.2.1 Arithmetic Operators . . . 33

4.2.2 Logical Operators . . . 36

4.2.3 Shift and Rotate Operators . . . 36

4.2.4 Relational Operators . . . 38

4.2.5 Selection Operators . . . 40

4.2.6 Unary Operators . . . 40

4.3 Parentheses . . . 41

5 Assembler Directives **43**

5.1 ASCII . . . 50

5.2 ASCIIC . . . 52

5.3 ASCIIZ . . . 52

5.4 ASCT . . . 53

5.5 BHS . . . 53

5.6 BLO . . . 53

5.7 BSCT . . . 54

5.8 BSS . . . 54

5.9 BTEXT 55
5.10 CHKSUM 56
5.11 CONC 57
5.12 CSCT 58
5.13 DSCT 58
5.14 EJECT 58
5.15 END 59
5.16 ENTRY 59
5.17 EQU 59
5.18 EQUREG (6809) 59
5.19 ETEXT 59
5.20 EXPORT 60
5.21 FCB 60
5.22 FCBS 61
5.23 FCBU 61
5.24 FCC 61
5.25 FDB 61
5.26 GOIF 62
5.27 GOIFZ 62
5.28 GOTO 63
5.29 GOULZ 64
5.30 GOUNLS 64
5.31 IMPORT 65
5.32 IMPRTB 66
5.33 INCLUDE 66
5.34 END 66
5.35 MACRO 66
5.36 MEND 66
5.37 MEXIT 67
5.38 MON 67
5.39 NAME 67

5.40 OPT 67
5.40.1 OPT ABS/REL 70
5.40.2 OPT CALIST/CANOLIST 70
5.40.3 OPT CMTCOL 71
5.40.4 OPT DB8/DB10/DB16 71
5.40.5 OPT DEBUG 71
5.40.6 OPT DIRECT/NODIRECT 72
5.40.7 OPT EOERR/NOEOERR 72
5.40.8 OPT FORMAT/NOFORMAT 72
5.40.9 OPT IGNORE/NOIGNORE 73
5.40.10 OPT INCLIST/INCNOLIST 73
5.40.11 OPT LBLLIMIT 73
5.40.12 OPT LINENUMBER 73
5.40.13 OPT LIST/NOLIST 74
5.40.14 OPT LLRANGE 74
5.40.15 OPT LONG/SHORT 75
5.40.16 OPT LPAGE 75
5.40.17 OPT MPERR/NOMPERR 75
5.40.18 OPT MPU 76
5.40.19 OPT MSHOW/NOMSHOW 76
5.40.20 OPT PAGENUM 76
5.40.21 OPT PARITY 76
5.40.22 OPT RV8/NORV8 77
5.40.23 OPT TABS 77
5.40.24 OPT TMERR/NOTMERR 77
5.41 ORG 78
5.42 PAGE 78
5.43 PRINT 78
5.44 PSCT 79
5.45 RCB 79
5.46 RCBS 79

5.47 RCBU 79
5.48 RDB 79
5.49 RMB 80
5.50 SBTTL 80
5.51 SBTTLE 80
5.52 SET 82
5.53 SETREG (6809) 82
5.54 SK1 82
5.55 SK2 83
5.56 SKIP1 83
5.57 SKIP2 83
5.58 SPACE 84
5.59 SPC 84
5.60 TEMPLT 84
5.61 TITLE 85
5.62 VERIFY 85
5.63 XDEF 86
5.64 XREF 86

6 Running the Assembler **87**

6.1 Getting Started 87
6.2 The Informed User 88
6.3 Run-Time Options 93
6.3.1 ABSOLUTE 96
6.3.2 CAMASK 96
6.3.3 COMMENT 97
6.3.4 CROSS 97
6.3.5 DEBUG 97
6.3.6 EOERR 97
6.3.7 ERROR 98
6.3.8 FFEED 98

6.3.9 HELP . . . 98
6.3.10 IGNORE . . . 99
6.3.11 INDENT . . . 99
6.3.12 LCA . . . 100
6.3.13 LINE NUMBER . . . 100
6.3.14 LIST . . . 100
6.3.15 LOC . . . 101
6.3.16 LPP . . . 101
6.3.17 MPerr . . . 101
6.3.18 MPU . . . 102
6.3.19 MSHOW . . . 102
6.3.20 PAGE . . . 103
6.3.21 RELOCATABLE . . . 103
6.3.22 REPORT . . . 103
6.3.23 SEQUENCE . . . 103
6.3.24 SLINE . . . 104
6.3.25 SRC . . . 104
6.3.26 SYMBOL . . . 104
6.3.27 TMERR . . . 104
6.3.28 TOC . . . 105
6.3.29 VALUE . . . 105
6.3.30 WIDTH . . . 105
6.4 After The Assembly . . . 105
6.5 Downloading Object Files from VAX/VMS . . . 106
6.6 Advanced Topics . . . 106
6.6.1 Parsing the Files and Options Specification Input String . . . 107
6.6.2 Alternative Modes of Assembler Execution . . . 108
6.7 Final Notes . . . 110

7 Conditional Assembly 111

8 Macro Assembly **117**

8.1 Macro Names 118

8.2 Macro Parameters 118

8.2.1 Positional Parameters 119

8.2.2 Formal Parameters 120

8.2.3 Parameter Substitution Strings 121

8.3 Macro Model Statements 123

8.4 String Concatenation 123

8.5 Automatic Label Field Generation 124

8.6 MEND 124

8.7 MEXIT 124

8.8 Macro Call 125

8.9 Nested Macro Calls 127

8.10 Hints for Using Macros 127

9 Relocatable Assembly **129**

9.1 Preliminary Discussion 129

9.2 Invoking Relocatable Assembly 130

9.3 Features of M6800 Relocatable Assembly 131

9.4 Implications of Relocatable Assembly 133

9.5 Example Usage of Relocatable Assembly 136

9.6 A Few Words About the S Record Format 144

10 Assembly Errors **147**

11 Summary of Nonstandard Features **161**

Chapter 1

Introduction

This manual is a user's guide for the Western Microsystems M6800 Family Assembler. For the sake of brevity, the rest of this manual refers to the Western Microsystems M6800 Family Assembler simply as the assembler or as the M6800 family assembler whenever one of the simpler terms can be used without ambiguity.

The assembler runs on any VAX computer system that uses the VMS operating system. The reader is expected to be somewhat familiar with the VAX/VMS environment, so this manual does not include explanations of simple VMS commands such as `PRINT` and `TYPE`. The reader is also expected to have some previous experience with assembly language and with the use of a normal two-pass assembler. The M6800 family assembler is a two-pass assembler, so it has many of the characteristics that are inherent in any two-pass assembler.

The M6800 family assembler features a run-time option that allows it to assemble programs for any of the following microprocessors: M6800, M6801, M6802, M6803, M6808, and H6301. The first five microprocessors in the family were all developed by Motorola, and they are manufactured both by Motorola and by other manufacturers who serve as second sources. The H6301 is a CMOS microprocessor that was developed by Hitachi as an extension of the M6801 architecture. All six of these microprocessors share the basic architectural features of the M6800 microprocessor, but the later processors in the family include new instructions and/or other features that aren't available with the M6800, which was the first processor in the family. Please refer to the appropriate microprocessor manual for detailed information regarding the architecture and instruction set of the particular microprocessor that you want to use.

The M6800 family assembler also features a run-time option that allows it to assemble programs for a 6502 microprocessor or for a GTE 65150 microprocessor. The 6502 is similar to the M6800 microprocessor in many respects, but it is considerably different in many other ways. The GTE 65150 microprocessor, which was designed primarily for use in telephone systems, is essentially an extension of the 6502 architecture. Please refer to the 6502 or 65150 microprocessor manual for detailed information regarding these processors.

By default, the assembler runs as a 6800 assembler, but a programmer can also tell the assembler to assemble for one of the other processors.

A conditionally assembled sister version of the M6800 family assembler is available to assemble programs for the M6809 microprocessor. The M6809 assembler is completely compatible with the M6800 family assembler except for minor differences that are dictated by the unique characteristics of the M6809 assembly language and its source syntax.

The M6800 family assembler includes several features that make source preparation and program documentation easier for the user. For example, the assembler has a text mode that can be used for an introductory comment block at the beginning of each logical section of a program. Also, the assembler has a variable comment column that allows a programmer to type comments (without the need for any special introductory comment characters) starting at or beyond the specified comment column. The assembler also has a useful feature that allows it to assemble source files that contain dot commands, which are commonly used by text formatters such as runoff, compose, prose, nroff, and troff, so the assembler can expediently assemble an intermediate version of a source file that will eventually be formatted by a text formatter in its final form.

The assembler accepts symbolic labels with as many as 80 characters per label, so programmers can employ meaningful labels that are reasonably descriptive. The assembler also accepts local labels. A local label can be defined over and over for many different local applications in a single program. Since local labels can be redefined, they are especially suitable for local uses in which normal symbolic labels aren't appropriate. Local labels are commonly used as destinations of local branches within a routine, and they serve other local purposes as well.

The assembler optionally generates an alphabetized concordance listing that lists each nonlocal label along with the line number of the statement that defined the label, the value of the label, and the line number of each reference to the label. The concordance listing is especially valuable for programmers who are developing or maintaining large programs with many variables and subroutines.

The assembler has a rich repertoire of operators that can be used in expressions. For example, besides recognizing the traditional arithmetic operators, the assembler also recognizes logical operators, shift and rotate operators, signed and unsigned relational operators, and a few miscellaneous operators. For users who require involved expressions, the assembler allows various styles of parentheses to force precedence as desired. Parentheses can be nested to virtually any level. When parentheses are not used to dictate precedence, the assembler simply evaluates expressions from left to right with all operators having equal precedence, so the user doesn't need to memorize any arbitrary precedence relationships among the many operators.

The assembler allows statement continuation from one source line to another, so a programmer can conveniently continue a single statement over any desired number of lines. The continuation feature gives the user added control over the source format and, consequently, over the listing format. For example, line continuation permits a user to break up long

statements that would otherwise ruin the uniform appearance of the listing by extending into the columns that are normally used for the comment field.

The assembler has an extensive set of assembler directives. Besides recognizing assembler directives that generate object values, reserve memory space, and/or define labels, the assembler also features directives for generating and formatting a table of contents, adding references to the concordance, and controlling several assembler options. Many of the assembler's options can be controlled either by option statements in the user's source file or by run-time options that can be dynamically specified from the terminal or job stream when the user runs the assembler.

The assembler recognizes an `INCLUDE` statement that includes a specified source file at a specified point in the assembly. The `INCLUDE` feature allows nested includes to any desired level subject only to the maximum number of files that the user's VMS account is allowed to have open at any one time. The VMS system manager can validate an individual user's account to permit the required level of include processing, and the assembler automatically provides the maximum level of nesting that is currently allowed for the account. The `INCLUDE` feature is especially valuable for large programs because it allows a user to structure a program as a tree of logical modules. The `INCLUDE` feature is also particularly useful to a chief programmer or a project manager who is managing a large project that includes many modules developed by several programmers.

One of the assembler's options allows it to run either as an absolute assembler or as a relocatable assembler. The assembler runs as an absolute assembler by default because most applications for the M6800 family require programs that reside in ROM at specified memory locations. However, the assembler can also run as a relocatable assembler and produce relocatable object modules that are suitable as inputs to the M6800 family linker. The user must then invoke the linker to link the desired relocatable object module(s) into an executable object module. When the assembler runs as an absolute assembler, it directly generates an executable object module that is in loader format. In this case, no linking step is necessary.

The assembler provides a powerful set of directives for conditional assembly. The conditional assembly can be conditional on items that are specified in the source file, and it can also be conditional on a value that the user can specify as a run-time option when invoking the assembler. The ability to make assemblies conditional on a value that is specified as a run-time option allows a user to generate multiple object versions of a program without editing the program's source file.

The assembler includes a flexible macro facility that allows a user to customize or extend the assembler's instruction set. The assembler's macro facility provides the following features: positional or formal parameters, default strings for formal parameters, nested macro definitions, nested macro calls, string concatenation, automatic label field generation, delimited or undelimited strings, continuation of undelimited strings, user-controlled listing or suppression of expanded lines, and controllable syntax error reporting that can be partially suppressed by the user.

Some assemblers and compilers aren't very helpful when they report errors. For example, some assemblers issue meaningless error numbers (such as "error 735") or cryptic, general-purpose error messages (such as the ever-popular "syntax error"). When the M6800 family assembler detects an error, however, it generates a specific, understandable error message that pinpoints the cause of the error for the user. In most cases the error message itself contains enough detail to eliminate any need for the user to refer to the user's manual for additional information.

Besides generating understandable error messages, the assembler also links error lines together into an error list. Below each statement with an error the assembler generates a linking message that specifies the line number of the previous statement with an error. The summary at the end of the listing specifies the line number of the last statement (if any) with an error, so a user can easily start at the end of an assembly and check each erroneous statement without bothering to examine the entire listing.

The assembler has one other notable feature that will quickly become apparent to the user. The assembler is fast. In benchmarks against other assemblers that run on VAX/VMS systems, the M6800 family assembler has consistently shown itself to be from 5 to 10 times as fast as the other assemblers. The speed differential varies according to the specific application, of course, but every test case has shown that the M6800 family assembler is significantly faster than other assemblers.

The chapters that follow describe the assembler's features in detail. The chapters (and the assembler) are organized in a way that allows a casual or novice user to skip some chapters entirely and still be able to use the assembler proficiently at the desired level of understanding. For example, a beginning user can easily start using the assembler without reading about conditional assembly or macros.

Chapter 2

Source Syntax

A source statement in assembly language for a microprocessor in the M6800 family can contain as many as four fields. These fields are known (from left to right) as the label field, the opcode field, the operand field, and the comment field. The following line is a typical source statement that illustrates all four fields as they would be typed into a source file.

```
LABEL   LDAA    CHAR, X         Get the next character into the AR.
```

Any combination of horizontal tab(s) and/or blank space(s) is known as white space, and the assembler normally recognizes white space as a field delimiter. The assembler also allows (but does not require) white space between the subfields of a multi-part field as illustrated by the space that appears between the comma and the `X` in the operand field of the example statement above. The assembler does not require white space after the last field of a line because the carriage return at the end of the line terminates the last field of the line.

The definition that was given above for white space isn't complete. Besides recognizing horizontal tabs and blank spaces as white space, the assembler also accepts a grave accent (`) as a white-space character, so white space actually consists of any combination of horizontal tab(s) and/or blank space(s) and/or grave accent(s). The assembler's recognition of a grave accent as a white-space character provides a benefit that relates to compatibility with text formatters such as runoff and compose. The advantages of using a grave accent as a white-space character in certain situations will be discussed later.

Most programmers use one or more tab characters to separate fields in a source line because standard tab stops conveniently align the source file (and ultimately the symbolic listing, if any) into neat columns. Tabs are generally better than blank spaces as field delimiters because field alignment is more difficult for anyone who uses blank spaces as delimiters. The assembler does have a `FORMAT` option that can provide automatic field alignment of the listing even if the source file is badly misaligned, but the `FORMAT` option is

intended for the special case in which a user needs to resurrect an antiquated source file that was originally produced in a primitive environment that didn't support tabs. The `FORMAT` option will be described in detail later. In the meantime, the reader is advised that tabs are universally preferred as field delimiters.

Although the assembler is relatively flexible regarding columnization, nearly all users adhere rigorously to standard columns as a matter of common practice. The label field always starts in column 1 as required by the assembler. By convention, the opcode field starts at the first tab stop, which is in column 9. Similarly, the operand field starts at the second tab stop, which is in column 17. Also by convention, the comment field starts at the fourth (not the third) tab stop, and the fourth tab stop is in column 33.

2.1 Source Characters

The assembler accepts a maximum of 126 characters in a single source line, and the assembler simply truncates any source lines that are too long. Most users automatically limit their source lines to fewer than 80 characters because lines that contain 80 or more characters are less convenient to edit. In fact, most users limit their source lines to a maximum of 75 characters per line because that line length is standard for several applications. Notice that a tab character is a single character in a source line, but a tab appears to occupy several columns of a line when the line is displayed or printed. A line that appears to exceed 126 columns might therefore be acceptable to the assembler without truncation if some of the characters in the line are tabs.

The assembler essentially ignores (i.e., does not process) delete codes and control characters other than tabs, but those invisible characters do count toward the limit of 126 characters per line just as tabs do. The assembler's policy of ignoring invisible characters prevents subtle problems that might otherwise occur for users who accidentally edit invisible characters into their source files.

Except where characters appear in quoted strings or as quoted ASCII literals in expressions, the assembler ignores the case of alphabetic characters. A user can therefore type labels, opcodes, and expressions in any desired mixture of uppercase and lowercase, and the assembler considers each lowercase character to be equivalent to its uppercase counterpart. Although the assembler is insensitive to case, programmers conventionally use pure uppercase for everything but comments. For comments, of course, people normally use an appropriate mixture of uppercase and lowercase just as this paragraph uses a mixture of uppercase and lowercase to enhance readability.

2.2 Comments

If the first column of a source statement contains an asterisk, the assembler considers the entire line to be a comment line. An asterisk in column one is therefore known as a comment indicator, and it provides one method for typing a comment. Better methods

of typing comments are available, though, and the assembler recognizes the asterisk comment indicator chiefly for purposes of backward compatibility to earlier assemblers. The BTEXT/ETEXT feature, which is described later, is best for blocks of full-width commentary such as the introduction that normally appears at the beginning of a routine or with the definition of a data structure, and comments that relate to short groups of instructions are best typed as automatic side comments, which are described in the paragraphs that follow.

If seventeen or more columns of white space occur at the beginning of a source statement, the assembler automatically considers the entire line to be a comment. Similarly, if seventeen or more columns of white space occur immediately after the label field, the assembler processes the label field and automatically considers the rest of the line to be a comment. Notice that the assembler conveniently counts columns of white space, not white-space characters. A tab character is therefore equivalent to an appropriate number of columns of white space depending on the position of the tab character in the source line, and the assembler's interpretation of a tab agrees with the user's intuitive feeling regarding columns of white space.

The number "seventeen" that appears in the paragraph above is not rigidly fixed. Instead, it is merely the standard default value for the parameter that is known as the assembler's CMTCOL parameter. If the default value of the CMTCOL parameter is unsuitable for some particular application, a programmer can easily set the CMTCOL parameter to a more appropriate value by using the "OPT CMTCOL=expression" assembler directive, which is explained in detail later. The assembler allows a user to change the CMTCOL value as often as desired in a single source program, but most programmers simply use the standard default CMTCOL value and never change it at all.

The CMTCOL feature allows a programmer to have an automatic comment field without the bothersome requirement to type an asterisk comment indicator in column one of each comment line. This feature is especially convenient for side comments that are being used to describe a short group of instructions because the comments frequently extend one or more lines beyond the last instruction line of the code segment. For example, the following two program segments are functionally identical, but the second segment takes advantage of the automatic comment feature while the first uses asterisk comment indicators.

```
        STAA    0, X            Store the maximum value (from the AR)
        DEX                   into the bottom entry of the list.   Then
        BRA     LOOP          decrement the list pointer, and branch back
*                             to the top of the outer sort loop to see if
*                             the list is now completely sorted.

        STAA    0, X            Store the maximum value (from the AR)
        DEX                   into the bottom entry of the list.   Then
        BRA     LOOP          decrement the list pointer, and branch back
                              to the top of the outer sort loop to see if
                              the list is now completely sorted.
```

The benefit of being able to omit the asterisk comment indicators from column one doesn't appear to be significant in the short example above. However, the ability to omit comment indicators from an entire program provides a considerable advantage for the person who must type the source file. Also, the program listing is cleaner and less cluttered without the unnecessary distraction of the asterisks in column one of each comment line. Finally, with an old-style assembler that requires comment indicators, a comment indicator that is accidentally omitted can cause an assembly error and require the programmer to waste time by editing and assembling again.

2.3 Label Field

The label field starts in column one (i.e., the first column), and white space in column one indicates that the label field has been omitted. A user who wants to omit the label field normally types a tab in column one to skip directly to the opcode field.

If the label field of a statement isn't blank, it can contain a symbolic label, a local label, or a conditional label. A symbolic label, commonly known simply as a label, is the ordinary kind of label that almost every programming language allows. A local label is a special kind of label that can be defined over and over for various local uses, and a conditional label (sometimes known as a system label) is a label that can be used as a target in conditional assembly. The paragraphs that follow discuss the three varieties of labels.

2.3.1 Symbolic Labels

The first character of a symbolic label in a normal application program must be an alphabetic character (`A`–`Z` or `a`–`z`). The assembler also recognizes a colon (`:`), an ampersand (`&`), or an underscore (`_`) as the first character of a symbolic label, but these special characters should not be used to start labels in normal programs. Any label that either starts or ends with a special character is reserved by convention for use with system software that is properly part of the operating system. A programmer who heedlessly starts or ends a label with a special character in a normal application program may therefore encounter serious (but richly deserved) difficulties.

Subsequent characters after the first character of a symbolic label can be alphabetic characters (`A`–`Z` or `a`–`z`), numeric characters (`0`–`9`), or special characters (`:`, `&`, or `_`). Although the special characters can be used freely as embedded characters within labels, a label in a normal application program should never end with a special character because any label that ends with a special character is reserved by convention for use with system software.

A symbolic label can be as short as one character or as long as 80 characters. By default, the assembler interprets and records a maximum of 80 significant characters per label, but a programmer can use the "`OPT LBLLIM=expression`" statement, which is described in another section, to tell the assembler to limit the number of significant characters per

symbolic label to any desired length in the range one through 80. The assembler effectively ignores any characters beyond the number of significant characters in a label.

The assembler allows a programmer to define and use any label that syntactically qualifies as a label, even if the label happens to match a macro name, an opcode mnemonic, or some other predefined mnemonic. Therefore, a user doesn't need to remember any arbitrary list of reserved words that can't be used as labels.

White space terminates the label field, and a user customarily types a single tab character to advance from the label field to the opcode field. A programmer who wants to define a label on a line by itself can use a carriage return to terminate the label field. If a label appears on a line by itself (or if `CMTCOL` columns of white space appear between the label and the next nonwhite character), the assembler defines the label to have the value of the location counter. A label that appears on a line by itself therefore receives the same value that it would have if an executable instruction appeared in the same statement with the label.

2.3.2 Local Labels

While a normal symbolic label can be defined only once in a program, a local label can be defined over and over any number of times. Local labels can therefore be used for local applications by several different sections of a single program, and various sections that define and use the same local label don't conflict with each other.

A programmer defines a local label by using a `jH` (j here) entry in the label field where `j` is any digit from 0 through 9, and a programmer can refer to a local label by writing `jB` (j backward) or `jF` (j forward) as a term of an expression. For example, suppose that a programmer defines the label `3H` in the label field of a statement. The programmer can refer to that label as `3B` in the operand field of an instruction that appears in the source program somewhere after the statement that defines the `3H` label, and the programmer can refer to the label as `3F` in the operand field of an instruction that appears somewhere before the statement that defines the `3H` label.

The use of a `jB` term in an expression refers to the closest previous definition of the `jH` local label, and the use of a `jF` term in an expression refers to the closest following definition of the `jH` local label. A `jB` or `jF` term in an expression never refers to a local label in the label field of the same statement that contains the `jB` or `jF` term, so a backward or forward reference always refers to a previous or following statement, never to the current statement.

A `jH` entry is valid in the label field, but a `jH` entry is not valid in an expression. Similarly, a `jB` or `jF` entry can be used in an expression, but a `jB` or `jF` entry is not valid in the label field as a label that is to be defined.

The following example shows two sequences of instructions that are functionally identical, but the sequence on the left uses symbolic labels while the sequence on the right uses local labels.

```
LBLA    EQU     *-29                    1H      EQU     *-29
        ORG     LBLA+35                         ORG     1B+35
LBLB    FDB     LBLC-*+LBLD             2H      FDB     3F-*+4F
LBLC    FCB     LBLD-LBLB               3H      FCB     4F-2B
LBLD    RMB     *-LBLB+LBLC-LBLA        4H      RMB     *-2B+3B-1B
```

The coding sequences in the next example illustrate the fact that the same local label can be defined over and over, and they also illustrate the fact that a `jB` or `jF` local label in an expression never refers to the `jH` label in the same statement. The two sequences of instructions are functionally identical, but the sequence on the left uses symbolic labels while the sequence on the right uses local labels.

```
LBLA    EQU     *-29                    3H      EQU     *-29
        ORG     LBLA+39                         ORG     3B+39
LBLB    FDB     LBLA-*+LBLC+LBLD        3H      FDB     3B-*+3F+1F
LBLC    FCB     LBLD-LBLB               3H      FCB     1F-3B
LBLD    RMB     *-LBLC                  1H      RMB     *-3B
```

The examples that are shown above are probably somewhat confusing because they are deliberately contrived to illustrate the subtle points of the rules regarding local labels. In actual practice, however, local labels aren't confusing at all. The following example illustrates a typical application of local labels. The code on the left uses symbolic labels of dubious value, and the analogous code on the right makes appropriate use of local labels.

```
        LDX     =LIST                           LDX     =LIST
        LDAA    =LENGTH                         LDAA    =LENGTH
        LDAB    KEY                             LDAB    KEY
        CLR     COUNT                           CLR     COUNT
BACK    CMPB    0, X                    1H      CMPB    0, X
        BNE     NEXT                            BNE     2F
        INC     COUNT                           INC     COUNT
NEXT    INX                             2H      INX
        DECA                                    DECA
        BGT     BACK                            BGT     1B
```

Local labels spare the programmer from the tiresome and unrewarding chore of inventing symbolic names for nearby locations when the programmer merely wants to transfer control to an instruction a few lines away. Programmers who don't use local labels usually end up using relatively meaningless symbolic labels like `LOOP`, `LOOP1`, and `LOOP2`, or `NEXT`, `NEXT1`,

and NEXT2. Occasionally a desperate programmer without access to local labels even resorts to symbolic labels like SAM, JOE, NANCY, and MARY.

As a general rule, an instruction label that doesn't have global significance to the program as a whole should probably be a local label. Variable names, of course, should ordinarily be meaningful symbolic labels.

A good programmer typically uses a normal symbolic label to define the entry point of a routine, and then the programmer uses local labels as local branch points within the routine. The same local labels can be used again as local branch points within the next routine with no conflict. As a matter of good style most programmers use local labels in ascending order starting with 1H (rather than 0H) in each routine to enhance readability, consistency, and maintainability, but the assembler doesn't enforce any such requirement.

The assembler has a special feature to guard against subtle logic errors that can occur when a user accidentally omits the definition of a local label. Suppose that a programmer refers to the 3F local label and then forgets to define it. With ordinary symbolic labels an omitted label definition would cause the assembler to report an error because of an undefined symbol. With local labels, however, the assembler is likely to find a 3H label in some subsequent routine, perhaps much later in the program, and then the 3F reference is unfortunately satisfied by the wrong label. To guard against this kind of devastating error, the assembler issues a warning message whenever a reference to a local label is too far away from the line that defines the local label.

The assembler considers a reference to a local label to be too far away from the local label's definition when the reference is more than 104 lines (about two full pages) away from the definition. The number 104 is merely the assembler's standard default limit, and a programmer can redefine the local-label separation limit as often as desired in a single program by using the "OPT LLRANGE=expression" directive, which is described in detail in another section. The distance between two source lines is defined to be the absolute value of the difference between their line numbers.

We can summarize the assembler's syntax rules for the definition of a local label as follows. If the first column of a statement contains a decimal digit, the assembler assumes that the programmer is using the label field to define a local label. The format for the definition of a local label requires an H (or an h) in column two, and column three must contain white space (or a carriage return) to terminate the label field. If columns two and three aren't formatted correctly when column one contains a decimal digit, the assembler reports an error in the format of the local label.

White space terminates the label field, and a user customarily types a single tab character to advance from the label field to the opcode field. A programmer who wants to define a local label on a line by itself can use a carriage return to terminate the label field. If a local label appears on a line by itself (or if CMTCOL columns of white space appear between the local label and the next nonwhite character), the assembler defines the local label to have the value of the location counter. A local label that appears on a line by itself therefore receives the same value that it would have if an executable instruction appeared in the same statement with the local label.

Local labels may initially be a little disquieting to a person who has never seen them before, but they soon become completely natural to anyone who uses them in a few programs. Most programmers who have used local labels wonder how anyone could ever get along without them.

2.3.3 Conditional Labels

If a dollar sign (`$`) appears in the first column of a statement, the assembler assumes that the label field contains a conditional label. A conditional label, which is also known as a system label, is used to label a target line in a conditional assembly. When the label field of a line contains a conditional label, the assembler ignores the rest of the line. Conditional labels and conditional assembly are discussed in detail in another section.

2.4 Opcode Field

The second field of a source statement in assembly language for a microprocessor in the M6800 family is the opcode field. The opcode field can contain one of the instruction mnemonics that is listed in the programmer's reference manual for the microprocessor, or it can contain the mnemonic for an assembler directive. The assembler directives, which are sometimes known as pseudo instructions, are all documented in another section of this manual.

Unlike some M6800 assemblers, the M6800 family assembler requires the user to type the entire opcode mnemonic as a single string with no embedded spaces. For example, while some assemblers accept "`LDA A`" and "`LDA B`" as valid opcodes, the M6800 family assembler requires `LDAA` and `LDAB` with no embedded spaces. The idea of allowing a space (as some assemblers do) between the basic instruction mnemonic and the character that specifies the accumulator sounds appealing at first, but it unfortunately introduces certain ambiguities. For example, consider the `CLR` instruction and the `CLRA` instruction. The `CLR` instruction clears the memory location that is specified by the operand field, and the `CLRA` instruction, which has no operand field, clears the microprocessor's AR (i.e., its "A" register or "A" accumulator). If the assembler allowed a programmer to use embedded spaces in an opcode mnemonic, then a "`CLR A`" statement could be either (1) a `CLR` instruction that clears a location that has the label `A` or (2) a `CLRA` instruction.

The `CLR` instruction is just one example of an entire class of ambiguities. Obviously, the assembler could resolve each ambiguity by imposing some arbitrary rules and restrictions, but special rules and restrictions are undesirable because they are unnatural and hard for users to remember. Therefore, the M6800 family assembler eliminates the ambiguities entirely by simply requiring the user to type the opcode mnemonic as a contiguous string with no embedded spaces. This single requirement is both natural and easy to remember.

Like the label field, the opcode field can be terminated by white space. A programmer ordinarily uses a single tab to advance from the opcode field to the operand field if the

instruction has an operand field. Some instructions (such as `INX` and `DEX`), however, don't have operand fields. For an instruction without an operand field, the opcode field can be terminated by a carriage return if the user doesn't want a comment field. More commonly, a programmer uses three tabs to advance from the opcode field of an instruction with no operand field directly to the comment field. The assembler knows which instructions don't have operand fields, of course, so even a single column of white space is sufficient to advance from the opcode field to the comment field for an instruction with no operand field. Nearly all programmers use three tabs in this case, though, to maintain proper field alignment for purposes of readability.

2.5 Operand Field

The third field of a typical statement is the operand field, which naturally contains the operand or operands for the instruction. Most instructions (e.g., "`STX SAM`" and "`ADDA JOE`") require at least a simple operand field, while other instructions (e.g., `RCB`, `RDB`, and the 6301's "`AIM =MASK, MEMORY`") require a minimum of two subfields in the operand field. Many instructions (e.g., "`LDAA 3, X`" and "`STAA 0, X`") optionally allow two subfields in the operand field, and a few instructions (e.g., the 6301's "`OIM =MASK, OFFSET, X`") allow three subfields in the operand field. Some instructions (e.g., `FCB`, `FDB`, `CONC`, and `OPT`) permit more than three subfields in the operand field. Finally, some instructions (such as `NEGA`, `SKIP1`, and `SKIP2`) don't require any operand field at all.

If an operand field contains two or more subfields, a comma must be used to terminate every subfield except the last one. The assembler allows (but does not require) white space after the comma that terminates one subfield and before the beginning of the next subfield. This rule makes the assembler's source syntax compatible with a standard typing rule that requires a blank space after a comma.

The operand field can be terminated by white space or by a carriage return. Most users type one or two tabs as necessary to advance from the operand field to the comment field with correct field alignment.

2.6 Comment Field

The fourth and last field of a typical statement is the comment field. The comment field can contain anything, and the assembler essentially ignores the comment field except for including it in the program's symbolic listing. The user should note, however, that string substitution can occur in the comment field during macro expansion, a feature that is more fully explained in the chapter on macros.

Although the assembler doesn't require or examine the comment field, complete comments are critically important to the quality of any program. Most users soon become aware of the value of good documentation, so they make extensive use of the comment field.

Programmers normally group a few instructions together as a logical step, and then they write a paragraph of commentary to explain the purpose and function of the logical step rather than explaining individual instructions. A single paragraph of comments therefore continues across the comment fields of several consecutive lines.

The assembler considers a blank or empty line to be a comment line, so a programmer can conveniently use a blank line to separate one logical group of instructions from the next logical group of instructions. The following example is only a brief fragment of a program, but it illustrates the method for grouping instructions into logical steps and explaining the logical steps.

```
        NEGA                    Set the next DELTA_T value in the list
        STAA    DELTA_T, X      equal to its original value minus the value
                                that we'll use for the new entry's DELTA_T
                                value. This step is necessary to account
                                for the new DELTA_T value that will be
                                inserted into the priority list.

        STX     NEWENT            Save a pointer to the position for the
        LDX     LISTEND         new list entry, and initialize the XR to
                                point to the location following the end of
                                the list. We want to move the tail of the
                                list down to make room for the new entry.

2H      DEX                       This loop moves the tail of the list down
        LDAA    0, X            by one entry to make room for the new
        STAA    ENTSIZE, X      entry. We move one byte at a time instead
        CPX     NEWENT          of moving an entire entry at a time because
        BNE     2B              we don't have a register that's big enough
                                to hold an entire entry.
```

Good programmers organize their comments as well-structured paragraphs of complete and proper English sentences with no arcane terms or cryptic abbreviations. The only intended audience for a comment is a human being, so programmers should always write comments with the same quality and style that are appropriate for any report, document, or article that humans will read.

2.7 Statement Continuation

The assembler allows statement continuation from one source line to the next, so a programmer can conveniently continue a single statement over any desired number of lines. Since the assembler's continuation feature permits a user to continue a statement over an

unlimited number of lines, programmers don't need to worry about any arbitrary restrictions on the total length of a statement.

The continuation feature gives the user added control over the source format and, consequently, over the listing format. For example, statement continuation allows a user to break up long statements that would otherwise ruin the uniform appearance of the listing by extending into the columns that are normally used for the comment field.

A hyphen (`-`) followed by either white space or a carriage return is known as a continuation indicator. Wherever white space is allowed in the noncomment part of a statement, a programmer can use a hyphen followed by either white space or a carriage return to indicate that the noncomment part of the statement continues with the first nonwhite character of the next line. A user can therefore break a statement and continue it to the next line following the label field, following the opcode field, or between any two subfields of the operand field.

A programmer can also specify statement continuation in an expression by typing a hyphen followed by either white space or a carriage return at any logical break in the expression. A logical break occurs in an expression immediately before and after every term, immediately before and after every operator, and immediately before and after every parenthesis. A single expression can be continued over as many lines as desired, so the length of an expression is unlimited.

The assembler does not allow statement continuation in the middle of a token where a token is a label, an opcode, a single term of an expression, an operator in an expression, a quoted string, a delimited macro argument, or a keyword such as an option mnemonic. The assembler also disallows statement continuation in the comment field (or anywhere after the end of the last field that precedes the comment field). Continuation of the comment field is unnecessary because the assembler's `CMTCOL` feature, which was explained earlier, effectively makes the continuation of comments totally automatic.

Incidentally, the `CMTCOL` feature doesn't apply to the white space (if any) at the beginning of a line that is a continuation of the previous line. The assembler's `CMTCOL` feature applies only at the beginning of a statement, not at the beginning of a line that is merely a continuation of a statement. A programmer can therefore indent as far as desired at the beginning of a continuation line, and the conventional practice is to indent each continuation line to the standard starting column for the field that is currently being typed.

If white space (rather than a carriage return) immediately follows the hyphen continuation character, then the assembler considers the rest of the line following the white space to be a comment. The significant part of the statement continues with the first nonwhite character of the next line. The first nonwhite character of the next line can even be another hyphen followed by white space, in which case the statement continues again to the next line.

Most programmers use statement continuation with long operand fields to prevent the operand field from extending into the columns that are normally used for the comment field. Consider the following examples:

```
LABEL   LDX     =TABLE+(ENTRY_SIZE*ENTRY_NUMBER)+FIELD_OFFSET-ADJUSTMENT

LABEL   LDX     =TABLE+-          The comment field is independent of line
                (ENTRY_SIZE*-   continuation, so programmers who use line
                ENTRY_NUMBER)+- continuation to continue long statements
                FIELD_OFFSET--  can conveniently type their comments just
                ADJUSTMENT      as they would ordinarily type them.

LABEL   LDX     =TABLE-           This statement, like the ones above, sets
                +(ENTRY_SIZE-   the XR to point to an individual byte of a
                *ENTRY_NUMBER)- particular field of a specified entry of a
                +FIELD_OFFSET-  sequential table. Each entry in the table
                -ADJUSTMENT     contains ENTRY_SIZE bytes.
```

The three statements above are all functionally equivalent to each other. The first statement doesn't use the continuation feature, and its operand field extends into the columns that are normally used for comments. The second statement uses the continuation feature to break the expression after each operator, and the third statement uses the continuation feature to break the expression before each operator. The second statement in the example above illustrates the format that most users seem to prefer for continued expressions, but the assembler has enough flexibility in this area to allow for a wide variation of individual preferences.

Notice that the assembler can tell the difference between a minus sign and a hyphen continuation character by the context in which the character appears. White space or a carriage return immediately follows a hyphen continuation character, but the character immediately following a minus sign is the next character of the expression or a hyphen continuation character. In any case, the character immediately following a minus sign is a nonwhite character.

Notice also that a user must not insert white space immediately before a hyphen continuation character in an expression because white space preceding the hyphen would normally terminate the expression. In general, white space terminates an expression except in the case of a white-space character that appears in an expression as a quoted ASCII character. White space can, of course, appear before a hyphen continuation character that is used to break a statement between two fields such as the label field and the opcode field or between two subfields of the operand field.

The following examples illustrate some more useful applications of line continuation:

```
LABEL1  FDB     EXPR1, -          This example shows a convenient method
                EXPR2, -        for continuing an instruction that contains
                EXPR3           multiple subfields in its operand field.
```

```
LABEL2  FDB     EXPR1,-          This example illustrates the fact that a
                EXPR2,  -        user can include any amount of white space
                EXPR3,     -     (including none) before a hyphen that
                EXPR4,        - appears between two fields or between two
                EXPR5            subfields of the operand field.

UNUSUALLY_LONG_LABEL_STRING -    This example illustrates the use of line
        LDAA    SAM              continuation to attach a long label to an
                                 instruction.

        MACRO                    This example shows how line continuation
        MOVE    SOURCE, -        can be useful when a person wants to define
                DESTINATION      a macro that has multiple arguments. The
        PSHA                     macro in the example is named MOVE, and it
        LDAA    SOURCE           has two arguments, SOURCE and DESTINATION.
        STAA    DESTINATION      Each argument is expected to be a string
        PULA                     that represents a valid operand field for
        MEND                     the instruction with which it is used.

        MOVE    <BASE, X>, -     This example uses line continuation in a
                TABLE+OFFSET     statement that calls the MOVE macro that
                                 was defined in the previous example.
```

The simple examples that are shown above are not, by any means, intended to be comprehensive. Indeed, they barely begin to illustrate some of the flexibility that is possible with the assembler's continuation feature, but they should be enough to give the reader a good idea of how to use statement continuation.

2.8 Assembler Support for Text Formatters

Programmers and other people who work with computers typically use automatic text formatters (such as runoff, compose, prose, nroff, or troff) to format nearly everything they write. A text formatter is an extremely useful program that formats text according to standard rules and/or formatting commands to improve both the appearance of the text and the consistency of its format. In terms of format and appearance, a text formatter essentially converts a rough draft into a final copy, so a text formatter helps a person produce neat, formatted text quickly and with a minimum amount of manual effort. Besides reducing manual effort, a text formatter also provides flexibility and generates output that is superior to the output that a person without a text formatter could expect to produce in a reasonable amount of time. Additionally, a text formatter encourages people to edit clarifications and other improvements into their text because the text formatter painlessly reformats the text and produces a new final copy that incorporates the improvements. The writer doesn't need

to worry about things like margins or pagination because the text formatter takes care of those details automatically.

A source program in assembly language is essentially a text file that has been carefully prepared in a particular format. Some parts of a source program contain assembly-language instructions that have a distinctive, columnized format, and other parts of a source program contain blocks of comments with the same kind of format that people use for ordinary text such as the text in a report or a business letter. A good text formatter can easily handle the columnized instruction format as well as the normal text format, so programmers might well consider the possibility of using text formatters to format their source programs.

Although a text formatter can easily provide the columnized format that is customary for assembly-language instructions, most programmers format their instructions manually while they type them because they can easily tab from one field to the next. Most programmers also format their comments to one degree or another while they type them because many kinds of formatting are quick and easy with a modern screen-oriented editor such as the E editor that is available on a VAX/VMS system. Some kinds of formatting are done most easily by text formatters, though, and a number of programmers use text formatters to polish the final appearance of their source programs. For example, a programmer might use a small portion of a text formatter's power simply to justify (i.e., align) the right margin in all of a program's comment blocks, and a programmer might also use a text formatter to center headings or to format tables.

2.8.1 Dot Commands

A person who wants to use a text formatter to improve the format of a source program usually needs to include a few formatting commands in the program's source file. Formatting commands might be necessary, for example, to set the margins, to turn justification on and off, and to control other options that the text formatter might have.

Formatting commands are commonly known as dot commands because nearly every text formatter in widespread use recognizes a formatting command by the fact that it starts with a period in column 1. The use of a period in column 1 to indicate a formatting command has become a de facto standard because a period seldom occurs at the left margin in ordinary text.

A programmer commonly includes dot commands in a source program with the intention of using a text formatter to justify margins and to provide other enhancements before printing a final listing of a frozen or released version of the program. However, the user might assemble the program several (perhaps even many) times during program development before the program is ready to be frozen, and the text formatter provides little or no benefit for these intermediate assemblies. In fact, a programmer who uses a text formatter to justify margins for every intermediate assembly is clearly wasting valuable computer time that could be better used for more worthwhile purposes such as processing a payroll or playing Adventure.

The assembler has a time-saving feature that allows a user to bypass the formatting step and to assemble a program that contains dot commands. The assembler recognizes a dot command by the fact that it starts with a period in column 1, and the assembler considers dot commands to be special kinds of comment lines. By default the assembler suppresses dot commands from the assembly listing, but it can optionally print them as comment lines when desired. Since the assembler normally suppresses dot commands from the assembly listing, the listing appears essentially as it would appear if the dot commands didn't exist in the source file. A programmer can therefore include dot commands in a source program to prepare it for final formatting, but the programmer can still assemble the program directly without wasting time to format the source program for intermediate assemblies.

2.8.2 Grave Accents

When a text formatter processes and reformats text, a single blank space in the original source text might change into multiple spaces or even a line break in the formatted output. For example, a space between words can change into a line break when a text formatter moves words from one line to another within a paragraph to produce lines with relatively uniform lengths. This process is known as filling the text. A single blank space can also change into multiple spaces if a text formatter inserts extra spaces between words to justify or align the right margin of a paragraph.

While most blank spaces are simply word separators that can be freely adjusted as described above, a few blank spaces are different and should not be modified or expanded. For example, suppose that a person is writing about Elizabeth I, who was the queen of England from 1558 until 1603. Imagine the momentary confusion that might result for a reader if a text formatter happened to change the space after "Elizabeth" into a line break. Generally speaking, a person's name (e.g., Thomas Jefferson) or a date (e.g., 4 July 1776) should not be split across a line or even expanded. Similarly, a range such as "A – Z" or "1981 – 84" shouldn't be split in a way that causes a hyphen to be the first character of a line. As another example, consider the blank spaces in a simple equation such as `a + b = c`. A text formatter that indiscriminately changed the space after the `a` into a line break would effectively mutilate the equation's readability.

Nearly every text formatter allows a user to specify some seldom-used special character as a nonexpandable space (i.e., as an indication of a single blank space that can't be changed or expanded). For example, a grave accent (`) almost never occurs in English text, so a person who is writing in English often includes a dot command to tell the text formatter that it should consider a grave accent to be a nonexpandable space. Then the user can safely type "Elizabeth`I" or "a`+` b`=`c" knowing that each grave accent will be translated into a single space in the text formatter's output.

The assembler conveniently classifies grave accents along with tabs and blank spaces as white-space characters. This esoteric feature is completely transparent to many users, but it is often helpful for programmers who use text formatters to format their source files. If a programmer uses grave accents as nonexpandable spaces for a text formatter, then the

programmer doesn't need to waste time formatting the source file for every intermediate assembly. Instead, the programmer can skip the formatting step altogether and submit the raw source file directly to the assembler, and the assembler interprets the grave accents in the raw source file as white-space characters. Some programmers use grave accents as nonexpandable blanks between the subfields of multi-part operand fields, and the assembler correctly recognizes these grave accents as white-space characters.

Chapter 3

Addressing Modes for Executable Instructions

The operand field of an executable instruction often specifies not just the location or value of the operand but also the addressing mode that the instruction should use to access the operand. The addressing modes that are available in the M6800 family are known as inherent addressing, immediate addressing, indexed addressing, extended addressing, direct addressing, and self-relative addressing.

3.1 Inherent Addressing

Several instructions (e.g., `ABA`, `CLRA`, `PSHB`, `DAA`, and `SWI`) are said to use inherent addressing. The instructions in this group are unique among executable instructions in that they don't have operand fields. In some cases, an instruction in this group (such as `WAI`) simply doesn't use an operand, and the lack of an operand obviously explains the lack of an operand field for the instruction. In other cases, an instruction in the inherent group uses one or more operands, but the instruction's operand(s) are inherently specified in the definition of the instruction itself. For example, the `ABA` instruction adds the BR to the AR, so its operands are inherently defined to be the BR and the AR. Similarly, the `CLRA` instruction clears the AR, so its operand is inherently defined to be the AR. The instructions that use inherently defined operands don't have operand fields because their operands are implicitly specified by their opcodes.

3.2 Immediate Addressing

Many instructions allow the use of a mode that is commonly known as immediate addressing. With immediate addressing, the actual value of the instruction's operand appears immediately after the instruction's opcode in memory. Instead of being a variable, the operand is a constant value that is known at assembly time (or perhaps at link time), and a

programmer types the operand's value rather than its address in the instruction's operand field.

A user who wants to specify immediate addressing must use either an equality sign (`=`) or a number sign (`#`) as the first character of the operand field. The assembler recognizes either of these characters as a request for immediate addressing, and the expression that immediately follows the equality sign or the number sign specifies the constant value of the instruction's operand. For example, "`LDAA =5`" loads the constant value 5 into the AR, and "`SUBB =47`" subtracts the constant value 47 from the BR. Similarly, "`LDX =3`" loads the value 3 into the XR, and "`CPX =0`" compares the XR against the value zero.

Some instructions (such as `LDX` and `LDS`) require 16-bit operands, and other instructions (such as `ADDA` and `CMPB`) use 8-bit operands.

When a programmer specifies immediate addressing with an instruction that uses a 16-bit operand, the assembler accepts any 16-bit value for the immediate operand. A programmer can therefore specify any expression value as long as that expression value can be represented in a 16-bit number system. A programmer can consider an immediate value to be either signed or unsigned, so the assembler accepts both signed and unsigned immediate operand values. For example, "`LDX =65535`" and "`LDX =-1`" are two ways of writing the same instruction because the unsigned value 65,535 is the same 16-bit number as the signed value −1.

When a programmer specifies immediate addressing with an instruction that uses an 8-bit operand, the assembler checks the expression value and reports an error for any immediate operand value that can't be correctly represented in an 8-bit field. A programmer can consider an 8-bit value to be either signed or unsigned, so the assembler accepts both signed and unsigned immediate operand values. A signed 8-bit value lies in the range from −128 through +127, and an unsigned 8-bit value lies in the range from zero through 255.

3.3 Indexed Addressing

Indexed addressing, which is available for many instructions, provides flexibility by allowing a user's program to compute an operand address dynamically at execution time. With indexed addressing the microprocessor computes the memory address of the instruction's operand by adding the current 16-bit value of the XR (i.e., "X" register or index register) to the unsigned 8-bit offset or base address that is specified in the instruction. Since the offset is an 8-bit unsigned value, the operand can reside in memory at any location in the range from `XR+$0000` through `XR+$00FF` where "XR" denotes the value of the XR and where a leading `$` indicates a hexadecimal number. The XR, of course, can contain any 16-bit value at execution time, so an instruction that uses indexed addressing can potentially access any memory location in the microprocessor's entire memory space.

A programmer who wants to use indexed addressing must include two subfields in the operand field. The first subfield contains an expression that specifies the desired 8-bit unsigned offset or base address, and the second subfield contains an `X` (or `XR`) to indicate

indexed addressing. For example, a user might write "`SUBA FIELD, X`" or "`ANDB 5, X`" to specify indexed addressing.

3.4 Extended Addressing

Extended addressing provides a 16-bit address field, so an instruction that uses extended addressing can conveniently access an operand anywhere in the microprocessor's entire memory space. The operand field for an instruction with extended addressing contains an expression that specifies the memory address of the desired operand. For example, "`INC COUNT`" uses extended addressing to increment the contents of a memory location named `COUNT`, and "`STX $1234`" uses extended addressing to store the 16-bit value of the XR into memory locations `$1234` and `$1235` with the most-significant byte of the XR value being stored into location `$1234` and with the least-significant byte of the XR value being stored into location `$1235`. The assembler always recognizes a leading `$` to indicate a hexadecimal number.

3.5 Direct Addressing

A programmer can use direct addressing with some instructions to access operands that reside in page zero of memory where page zero of memory is defined to be the 256 bytes from location `$0000` through location `$00FF`. Since the top 8 bits of a direct address are always zero by definition, direct addressing requires only an 8-bit field to specify the address of an operand. An instruction that uses direct addressing is therefore one byte shorter than the corresponding instruction with extended addressing, and an instruction that uses direct addressing consequently executes more quickly than the same instruction with extended addressing. Clearly, direct addressing offers some important advantages over extended addressing, but direct addressing can be used only for operands in page zero whereas extended addressing can be used for operands throughout the microprocessor's memory space.

The operand field for an instruction with direct addressing contains an expression that specifies the page-zero memory address of the operand, so the source syntax for direct addressing is essentially no different from the source syntax for extended addressing. For example, an instruction such as "`LDAB FLAG`" could use either direct or extended addressing since the source syntax in this case is the same for both addressing modes. The assembler must choose between direct and extended addressing based on the value of the expression and other considerations as explained below.

The assembler automatically selects direct addressing instead of extended addressing if (1) direct addressing is available for the particular instruction that is specified in the opcode field and (2) the expression in the operand field specifies an operand address in page zero and (3) the expression in the operand field doesn't contain any forward references. A forward reference is a symbolic label (or a `jF` local label) that hasn't already been defined to have a particular value by a previous statement in the source program.

The assembler must establish the length or size of each instruction during pass 1 of the assembly, so the assembler must choose between direct and extended addressing during pass 1. If an operand expression contains a forward reference, the assembler obviously can't evaluate the expression completely during the first pass of the assembly. Therefore, the assembler can't reliably determine (during pass 1) whether or not an operand with a forward reference resides in page zero, and the only safe course for an operand expression that contains a forward reference is to choose extended addressing. If the operand with the forward reference later turns out to be in page zero, the extended addressing is slightly wasteful but at least works correctly. Consider the alternative. If the assembler had chosen direct addressing during pass 1 and the operand with the forward reference later turned out to reside somewhere outside page zero, the instruction couldn't address the desired operand and therefore wouldn't even work.

At first glance, forward references might seem to be a potential source of significant irritation since they can possibly cause the assembler to select extended addressing when direct addressing would work more efficiently. This undesirable situation seldom occurs in actual practice, however, because forward references almost never appear in operand expressions for instructions that require the assembler to choose between direct and extended addressing. The kinds of instructions that force the assembler to choose between direct and extended addressing usually address either variables or constants, and good programmers don't make any forward references to variables or constants because they always define all of their variables and constants before they write any code that refers to those variables and constants.

Many compiler languages actually require programmers to define their variables and constants before they write any code, so programmers who have used compiler languages naturally define their variables and constants first by force of habit. Besides eliminating unwanted forward references, the practice of defining variables and constants at the beginning of a program enhances program readability by putting the foundation for the code at the front of the listing where it belongs.

A perverse programmer who insists on using direct addressing with an instruction that contains a forward reference can force the assembler to select direct addressing by including an exclamation point (!) as the first character of the operand field. The exclamation point tells the assembler that the user wants direct addressing regardless of any forward references. For example, "`STAB !TOTAL`" uses direct addressing even if `TOTAL` is a forward reference.

3.6 Self-Relative Addressing

A branch-class instruction in the M6800 family uses self-relative addressing to specify its destination address. Instead of containing an actual 16-bit destination address, a branch-class instruction contains a signed, 8-bit, self-relative offset value that can be used to determine the desired destination address. The microprocessor computes a self-relative destination address by sign extending the signed 8-bit offset to 16 bits and then adding the sign-extended offset value to the 16-bit value of the PR. The PR, which is also known as the

PC or the program counter, is a register that always points to the next sequential byte in the instruction stream. When the PR is being used to compute the self-relative destination of a branch-class instruction, it points to the byte that immediately follows the second byte of the branch-class instruction in memory.

Obviously, the signed 8-bit offset that appears in the second byte of a branch-class instruction must equal the difference between the desired branch destination and the PR value. The value of the offset is therefore `DEST` $-$ `PR` where `DEST` represents the desired destination address and `PR` represents the relevant value of the PR. Since the relevant PR value is the memory address of the byte that immediately follows the second byte of the branch-class instruction in memory, the value of the offset can be written as `DEST` $-(*+2)$ where the symbol $*$ represents the memory address of the first byte of the branch-class instruction.

A programmer who wants to use a branch-class instruction simply specifies the desired destination address, and the assembler automatically computes the self-relative offset value that will result in a branch to the indicated destination address at execution time. Therefore, the operand field for a branch-class instruction should contain an expression that directly specifies the desired destination address. For example, a user might say "`BGT LOOP`" to branch to location `LOOP` when the result of some operation is greater than zero. Similarly, a user might say "`BNE ERROR`" to branch to location `ERROR` if the condition codes don't indicate equality.

Note carefully that the programmer always specifies the desired destination address and that the assembler performs the menial computations that are necessary to generate the self-relative offset that the microprocessor uses. The programmer never needs to compute the offset value manually since the assembler automatically handles that chore.

Chapter 4

Expressions

A typical instruction such as "`LDAA KEY`" or "`STX POINTER`" has an operand field that contains an expression. Most expressions that programmers use are extremely simple expressions. In fact, in the large majority of cases, an expression contains only one term such as the label `KEY` or the label `POINTER`, but an expression can be significantly more complex. This chapter explains the rules that govern expressions.

The processors in the M6800 family have 16-bit address spaces, so the assembler uses 16-bit arithmetic to evaluate expressions. An expression can be simply a single term, or it can contain multiple terms connected by binary operators. An expression can also include a unary operator that applies to the first term of the expression.

The assembler evaluates expressions from left to right with no operator precedence rules, but a programmer can use various styles of parentheses to dictate any desired order of evaluation. An expression that appears within parentheses is called a subexpression, and the assembler treats a parenthesized subexpression as a single term of the expression that contains it. The same rules that apply to expressions also apply to parenthesized subexpressions.

4.1 Terms

An expression always contains one or more terms, so a term is a basic component of any expression. Table 4.1 summarizes the kinds of terms that the assembler accepts, and the following paragraphs explain the various kinds of terms in detail.

4.1.1 Binary Numbers

A `%` prefix character followed by a series of one or more bits (0s and 1s) is a binary number. A binary number can contain as many as 16 significant bits, so a user can specify full 16-bit values in binary if the need arises. The smallest acceptable binary number is `%0`, and the largest acceptable binary number is `%1111111111111111`.

Table 4.1: Summary of Terms

Type of Term	**Example**	**Remarks**
Binary Number	`%01000101`	Starts with `%`
Octal Number	`0123456`	Starts with zero
Decimal Number	`53147`	Starts with any nonzero digit
Hexadecimal Number	`$FEED`	Starts with `$`
Symbolic Label	`RESULT`	Any valid label
Local Label (Forward)	`3F`	`jF` for $0 \leq$ `j` ≤ 9
Local Label (Backward)	`5B`	`jB` for $0 \leq$ `j` ≤ 9
Quoted ASCII Character	`"D"`	ASCII value of character
Present-Location Symbol	`*`	Current memory address
RAD-40 Hash String	`?ABC`	16-bit RAD-40 hash code
Conditional-Assembly Mask	`.CAMASK.`	Run-time `CAMASK` value
Subexpression	`(ENTRY+LINK-1)`	Parenthesized subexpression

4.1.2 Octal Numbers

Any series of one or more octets (0–7) starting with a zero is an octal number. The assembler accepts octal numbers with as many as 16 significant bits because users sometimes find it convenient to code 16-bit constants as octal values. The smallest acceptable octal number is `0`, and the largest acceptable octal number is `0177777`.

4.1.3 Decimal Numbers

Any series of one or more decimal digits (0–9) starting with a nonzero digit is a decimal number. The assembler accepts decimal numbers in the range from 1 through 65,535. The number zero is technically an octal number, but users commonly think of it as a decimal number. In any case, the value zero is the same for any base.

4.1.4 Hexadecimal Numbers

A `$` prefix character followed by series of one or more hextets (0–9, A–F, or a–f) is a hexadecimal number, commonly known as a hex number. Hex numbers, like numbers in

the other bases, can contain as many as 16 significant bits, so a hex value can contain as many as four significant hextets. The smallest acceptable hexadecimal number is `$0`, and the largest acceptable hex number is `$FFFF`.

4.1.5 Symbolic Labels

Like any other term, a symbolic label has a 16-bit value. If the assembler is performing an absolute assembly, every label that appears as a term in an expression must be defined somewhere in the program. For a relocatable assembly, the assembler also accepts a label as a term in an expression if (1) the label is declared as a label that is defined in another module and (2) the context in which the expression is used permits a value from another module.

Some assembler directives have operand fields or subfields that must be evaluated accurately during pass 1, so the expressions that appear in these particular fields or subfields must not contain any forward references. For example, `ORG`, `EQU`, `BSS`, and `RMB` statements prohibit forward references. If a label that appears as a term in an expression hasn't already been defined to have a particular value when the assembler encounters the expression that contains the label, the label is being used as a forward reference. Therefore, any label that is initially defined on a subsequent line in the program is a forward reference. A label is similarly a forward reference (in the case of a relocatable assembly) if the label is declared as a label that is defined in another module.

Recall that the assembler allows a programmer to define and use any label that syntactically qualifies as a label, even if the label happens to match a macro name, an opcode mnemonic, or some other predefined mnemonic. For example, the instruction "`LDAA X, X`" has the label `X` in the first subfield of its operand field, and it has the predefined mnemonic `X` in the second subfield of its operand field. The assembler knows what belongs in each subfield, so it properly uses the label `X` as an offset value with indexed addressing, just as the programmer intended. In a related example, the assembler correctly interprets the instruction "`DEC DEC`" to be a `DEC` instruction that decrements the value at location `DEC`.

4.1.6 Local Labels

The assembler accepts a local label as a term of an expression if the statement that defines the local label isn't too far away from the expression that uses the local label. The assembler considers a local label's definition to be too far away from a reference to the local label when the definition is more than 104 lines (about two full pages) away from the reference. The number 104 is merely the assembler's standard default limit, and a programmer can conveniently redefine the local-label separation limit as often as desired in a single program by using the "`OPT LLRANGE=expression`" directive, which is described in detail in another section. The distance between two source lines is defined to be the absolute value of the difference between their line numbers.

A local label in an expression has the form `jB` (`j` backward) or `jF` (`j` forward) where `j` is a single digit in the range from 0 through 9. A `jB` local label in an expression refers to the closest previous definition of the `jH` (`j` here) local label, and a `jF` local label in an expression refers to the closest following definition of the `jH` local label. A `jB` or `jF` term in an expression never refers to a local label in the label field of the same statement that contains the `jB` or `jF` term, so a `jB` or `jF` term always refers to a label that is defined in a previous or subsequent statement. For example, the instruction "`7H BRA 7B`" refers to some previously defined `7H` label, not to the `7H` label in the same instruction. Similarly, the instruction "`4H BRA 4F`" refers to some `4H` label that will presumably be defined by a subsequent statement in the program.

A local label of the form `jF` is always a forward reference, so a `jF` term is illegal in an expression that is used in a context where forward references aren't allowed. For example, the expression value for an `ORG`, `EQU`, or `RMB` statement must be evaluated during pass 1, so these assembler directives don't allow forward references.

4.1.7 Quoted ASCII Characters

A single printable ASCII character that is enclosed by quotation marks or apostrophes (e.g., `"A"`, `'A'`) or merely preceded by a quotation mark or an apostrophe (e.g., `"A`, or `'A`) is always valid as a term of an expression. The 16-bit value that the assembler uses for the quoted character is the right-justified, zero-filled, 8-bit ASCII code of the character with the parity bit either set or reset according to the parity option that is currently selected. A programmer can easily specify the parity option (and change it as often as desired) by using the "`OPT PARITY=expression`" assembler directive, which is fully explained in the chapter that discusses assembler directives. If a programmer never uses an `OPT` instruction to specify the desired parity, the assembler uses zero parity by default. If zero parity is being used, then the 16-bit value of the term `"A"` (for example) is `$0041`. If mark parity is being used, on the other hand, then the 16-bit value of the term `"A"` is `$00C1`.

A user normally specifies a quoted ASCII character in an expression either by enclosing the character with quotation marks (e.g., `"A"`) or by enclosing it with apostrophes (e.g., `'A'`). The assembler accepts the funny-looking `"A` and `'A` forms primarily for the sake of maintaining compatibility with Motorola's original definition of the M6800 assembly language, and most programmers no longer use those forms.

4.1.8 Present-Location Symbol

When an asterisk (`*`) is used as a term in an expression, it represents the value of the assembler's location counter, which contains the 16-bit memory address of the instruction that is currently being assembled. The assembler updates the location counter as the last order of business when it assembles an instruction, so the value of an asterisk for any executable instruction is always the memory address of the first byte of the instruction. The value of an asterisk is similarly the address of the first byte of the object value for any

nonexecutable assembler directive that has a single operand field and generates an object value. If the instruction that is currently being assembled is an assembler directive that doesn't generate an object value, then the value of the location counter is the memory address that would have been used if the instruction had been an executable instruction. The value of an asterisk for a compound assembler directive such as `FCB` or `FDB` is the memory address of the first byte of the field that is currently being processed.

Although the asterisk certainly has valid uses, it can also be badly abused by poor programmers. For example, anyone who codes an instruction such as "`BRA *+5`" is making his or her program hard to read and hard to maintain. First of all, instruction lengths aren't necessarily one byte per instruction, so the destination of the branch isn't immediately obvious to a person who reads the program. Secondly, if any intervening code is inserted (or deleted) after the "`BRA *+5`" and before the target address of the branch, the program will no longer operate correctly. If the offending instruction is changed to a "`BRA 1F`" instruction, however, the program will be easier to read and will continue to perform correctly when intervening code is inserted (or deleted) between the branch and the destination of the branch.

4.1.9 RAD-40 Hash String

A question mark (`?`) immediately preceding a string of (usually three) label characters specifies a RAD-40 hash string as a term of an expression. Any character that is valid as a character in a label is also valid as a character in a RAD-40 hash string. The RAD-40 hash code is a compact encoding scheme that allows a user to encode three label characters into only two bytes instead of requiring three bytes as ASCII would, so a programmer can profitably use the RAD-40 code to compress storage requirements for strings of label characters.

Each label character has a RAD-40 code value in the range from 1 through 39, and the RAD-40 hash code for a string of three characters is the radix-40 number whose three base-40 "digits" correspond to the RAD-40 character codes for the three characters of the string. For example, the RAD-40 character codes for the characters `P`, `I`, and `G` are 28, 21, and 19, respectively, so the RAD-40 hash string `?PIG` specifies a term with the value $(28 * 40^2 + 21 * 40^1 + 19 * 40^0)$ or $(28 * 1600 + 21 * 40 + 19 * 1)$ or 45,659 or `$B25B`.

If a RAD-40 hash string contains fewer than three label characters, the assembler effectively left justifies the RAD-40 character codes in the radix-40 number to obtain the hash value of the shorter RAD-40 string. For example, the two-character RAD-40 hash string `?PI` specifies a term with the value $(28 * 40^2 + 21 * 40^1 + 0 * 40^0)$ or $(28 * 1600 + 21 * 40)$ or 45,640 or `$B248`. If a RAD-40 hash string contains more than three label characters, the assembler effectively ignores all characters beyond the first three.

Why does the assembler use radix 40 as the basis for an encoding scheme? The reason is simple. The largest possible three-place base-40 value is $(39 * 40^2 + 39 * 40^1 + 39 * 40^0)$ or $(39 * 1600 + 39 * 40 + 39 * 1)$ or 63,999 or `$F9FF`, so the value for any three-character RAD-40 hash string fits into a 16-bit field as an unsigned value. On the other hand, the

largest possible three-place base-41 value is 68,920 or `$10D38`, which exceeds the capacity of a 16-bit unsigned value. Radix 40 is therefore the largest radix we can use if we want to code a three-place value into 16 bits. Obviously, any encoding scheme needs one value (usually zero) to represent a null character, so we are left with the ability to encode a three-character string from any set of 39 characters into a 16-bit unsigned value.

The RAD-40 code values from 1 through 39 correspond to the label characters `&`, `0`–`9`, `:`, `A`–`Z`, and _, respectively. The RAD-40 code values for the lowercase alphabetic characters (a–z) are 13 through 38, the same as the RAD-40 code values for the uppercase alphabetic characters. A RAD-40 hash string is therefore insensitive to case, and we lose case information when we hash a string into its RAD-40 representation. Note that the collating sequence for RAD-40 strings is the same as the collating sequence for ASCII strings.

The RAD-40 code can be extremely useful for an application in which a user wishes to condense the storage requirements for strings of characters. However, the user must realize that the RAD-40 code applies only to label characters, not to all ASCII characters, and the user must further realize that the RAD-40 code does not preserve any case information. The RAD-40 code definitely has limitations, but it can be highly beneficial for a programmer who understands how to use it.

4.1.10 Conditional-Assembly Mask

The assembler accepts a special predefined token (`.CAMASK.`) as a term of an expression. The `.CAMASK.` term represents the 16-bit value of a conditional-assembly mask that can be specified as a run-time parameter when a user invokes the assembler, so a programmer has access to a parameter that can be dynamically specified as any desired value for a particular assembly. The chapter that explains how to run the assembler includes information about the conditional-assembly mask and the run-time option that sets the conditional-assembly mask for the current assembly, and the chapter that explains conditional assembly also contains information about the `.CAMASK.` feature.

The `.CAMASK.` term is frequently used with conditional assembly because the dynamic `.CAMASK.` value allows a programmer to change the assembly without editing and changing the program's source file. For example, a program that is written with conditional assembly might use the `.CAMASK.` value as a code that indicates something about the environment of the program's target machine, thus allowing a single source program to generate object programs for multiple target environments.

Although the `.CAMASK.` term is normally used with conditional assembly, the assembler accepts the `.CAMASK.` term in any expression. A programmer can therefore use the `.CAMASK.` term in any desired application. For example, a programmer might do something as simple as using the `.CAMASK.` value to specify the beginning memory address of the ROM area for which a program is being assembled, or a programmer could use the `.CAMASK.` term to specify the length of a list that must be allocated in a program's RAM area.

The `.CAMASK.` term is a special token, not a label, so references to the `.CAMASK.` term don't appear in the concordance. A programmer who wants the concordance to include

references to the conditional-assembly mask can equate a label to the `.CAMASK.` term and then use the label instead of using the `.CAMASK.` term. For example, a user might say "`CAMASK EQU .CAMASK.`" and then use the `CAMASK` label instead of using the `.CAMASK.` token.

4.1.11 Parenthesized Subexpression

A complete expression enclosed by parentheses is known as a parenthesized subexpression, and a parenthesized subexpression is valid as a term in an expression. A programmer can use parentheses to dictate the order of evaluation in an expression, and a programmer can also use parentheses to add clarity to an expression. Another section in this chapter explains the use of parentheses in detail.

4.2 Operators

The assembler has an extensive set of operators that can be used in expressions. For example, besides recognizing the traditional arithmetic operators, the assembler also recognizes logical operators, shift and rotate operators, signed and unsigned relational operators, and a few miscellaneous operators. Table 4.2 and Table 4.3 summarize the binary and unary operators, respectively. The following paragraphs explain each operator in detail.

4.2.1 Arithmetic Operators

The assembler recognizes binary arithmetic operators for addition, subtraction, multiplication, and division, and the assembler has provisions that allow a user to perform these operations with both signed and unsigned values.

The addition (`+`), subtraction (`-`), and multiplication (`*`) operators can all be used equally well with either signed or unsigned numbers. In each case the assembler retains the 16 least-significant bits of the result and discards any bits that carry beyond the 16 least-significant bits. The final 16-bit result of an addition, subtraction, or multiplication is the same for either a signed operation or an unsigned operation.

The assembler recognizes two division operators because the results of signed divisions and unsigned divisions are often different in ways that can't be reconciled by mere truncation. Similarly, the assembler accepts a signed and unsigned modulo operator.

The division operator (`/`) specifies a signed integer division, so this operator should be used only with values in the range from $-32,768$ through $+32{,}767$. The assembler simply discards any remainder from a signed integer division, and it truncates the signed quotient toward zero. For example, $(+8)/(+3) = +2$, $(+8)/(-3) = -2$, $(-8)/(+3) = -2$, and $(-8)/(-3) = +2$.

The `.DIVUNS.` division operator specifies an unsigned integer division, so the `.DIVUNS.` division operator should be used with unsigned values in the range from zero through

Table 4.2: Binary Operators for Assembler Expressions

Operator	**Remarks**
`+`	Add (signed or unsigned).
`-`	Subtract (signed or unsigned).
`*`	Multiply (signed or unsigned; 16-bit product).
`/`	Divide (signed integer division; remainder discarded).
`.DIVUNS.`	Divide (unsigned integer division; remainder discarded).
`.MOD.`	Modulo function for signed values.
`.MODUNS.`	Modulo function for unsigned values.
`.AND.`	Bitwise logical AND.
`.OR.`	Bitwise logical inclusive OR.
`.XOR.`	Bitwise logical exclusive OR.
`.ASL.`	Arithmetic shift left (16-bit arithmetic shift; zero fill).
`.ASR.`	Arithmetic shift right (16-bit arithmetic shift; sign fill).
`.LSL.`	Logical shift left (16-bit logical shift; zero fill).
`.LSR.`	Logical shift right (16-bit logical shift; zero fill).
`.ROL.`	Rotate left (16-bit rotate with no carry bit involved).
`.ROR.`	Rotate right (16-bit rotate with no carry bit involved).
`.GT.`	If signed expression $>$ signed term, then 1; otherwise, 0.
`.GE.`	If signed expression $\geq$ signed term, then 1; otherwise, 0.
`.EQ.`	If signed expression $=$ signed term, then 1; otherwise, 0.
`.NE.`	If signed expression $\neq$ signed term, then 1; otherwise, 0.
`.LE.`	If signed expression $\leq$ signed term, then 1; otherwise, 0.
`.LT.`	If signed expression $<$ signed term, then 1; otherwise, 0.
`.HI.`	If unsigned expression $>$ unsigned term, then 1; otherwise, 0.
`.HS.`	If unsigned expression $\geq$ unsigned term, then 1; otherwise, 0.
`.EQ.`	If unsigned expression $=$ unsigned term, then 1; otherwise, 0.
`.NE.`	If unsigned expression $\neq$ unsigned term, then 1; otherwise, 0.
`.LS.`	If unsigned expression $\leq$ unsigned term, then 1; otherwise, 0.
`.LO.`	If unsigned expression $<$ unsigned term, then 1; otherwise, 0.
`.MIN.`	Select the smaller of two signed values.
`.MAX.`	Select the larger of two signed values.
`.MINUNS.`	Select the smaller of two unsigned values.
`.MAXUNS.`	Select the larger of two unsigned values.

Table 4.3: Unary Operators for Assembler Expressions

Operator	**Remarks**
`+`	Do not negate the following term.
`-`	Negate the following term.
`.BITS.`	Count the number of one bits in the following term.
`.COM.`	Unary one's complement of the following term. $0000 ⟷ $FFFF.
`.LOG2.`	Unary base-2 logarithm of the following term.
`.NOT.`	Unary logical not of the following term. True ⟷ False.

65,535. The assembler discards any remainder from an unsigned division, and it truncates the unsigned quotient toward zero. For example, `$FFFF.DIVUNS.2` = `$7FFF`, and `60000.DIVUNS.50000` = `1`.

The assembler's `.MOD.` operator performs the modulo function for signed values, so it should be used with values in the range from $-32,768$ through $+32{,}767$. The assembler defines the result of `<expression>.MOD.<term>` to be the signed remainder that results from dividing the signed value of the expression on the left by the signed value of the term on the right. For example, `(+5).MOD.(+3)` = $+2$, `(+5).MOD.(-3)` = $+2$, `(-5).MOD.(+3)` = -2, and `(-5).MOD.(-3)` = -2.

The assembler's `.MODUNS.` operator performs the modulo function for unsigned values, so it should be used with values in the range from zero through 65,535. The result of `<expression>.MODUNS.<term>` is the unsigned remainder that results from dividing the unsigned value of the expression by the unsigned value of the term. For example, `$FFFF.MODUNS.2` equals 1, and `2.MODUNS.$FFFF` equals 2.

Besides being used as binary operators, the addition and subtraction operators can also be used as unary operators as shown in the examples above for signed division. A programmer can include a unary operator at the beginning of an expression or parenthesized subexpression, and the unary operator applies to the first term of the expression or parenthesized subexpression. Since an entire parenthesized subexpression is a term, a programmer can use parentheses to make a unary operator apply to any desired part of an expression. For example, the expression $-(10 + 2)$ evaluates to -12.

An asterisk (`*`) has two possible meanings in an expression, but the syntax of an expression unambiguously indicates whether an asterisk is being used as the multiplication operator or as the present-location symbol. For example, the rather unlikely expression `***` represents the value of the location counter multiplied by the value of the location counter.

4.2.2 Logical Operators

The assembler recognizes three logical operators. The `.AND.` logical operator specifies a 16-bit logical AND operation, the `.OR.` logical operator specifies a 16-bit logical inclusive OR operation, and the `.XOR.` logical operator specifies a 16-bit logical exclusive OR operation. The assembler uses mnemonic tokens such as `.AND.`, `.OR.`, and `.XOR.` instead of using special characters for operators beyond the simple arithmetic operators because the mnemonic tokens are easier for people to understand and remember. For example, `.XOR.` is easy to understand and remember, but people might have trouble trying to remember a particular combination of special characters such as "`|+`" to specify an exclusive OR operation.

As a typical example of a good use of a logical operator, consider a situation in which a person wants to load the address (not the contents) of location `VECTOR` into the AR and the BR with the AR receiving the most-significant byte of the address of location `VECTOR` and with the BR receiving the least-significant byte of the address of location `VECTOR`. A programmer could write "`LDAB =VECTOR.AND.$00FF`" as an appropriate instruction to get the least-significant byte of the address of location `VECTOR` into the BR. The `.AND.` operation eliminates the most-significant byte of the address of location `VECTOR`, and it therefore leaves the least-significant byte of the memory address as the value of the expression.

4.2.3 Shift and Rotate Operators

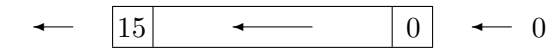
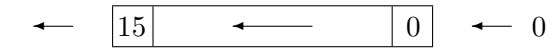
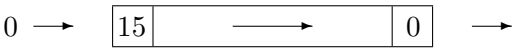
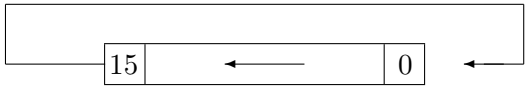
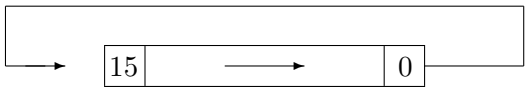
The assembler accepts operators for both arithmetic and logical shifts, and it also recognizes operators that specify rotations either to the left or to the right. In each case, the assembler uses the 16-bit value of the term immediately following the shift or rotate operator as an unsigned bit count that indicates the number of bit positions by which the current 16-bit expression value should be shifted or rotated. For example, `$5217.ROL.4` means to rotate the value `$5217` to the left by 4 bit positions, thus obtaining the value `$2175`. A bit count of zero is valid, but it always specifies a null shift or rotate and effectively does nothing.

Table 4.4 graphically illustrates the functions of the shift and rotate operators, and the following paragraphs explain the shift and rotate operators in detail. The discussions of the shift and rotate operators refer to the leftmost bit of a 16-bit value as bit 15, and they refer to the rightmost bit as bit zero.

The `.ASL.` operator specifies an arithmetic shift to the left. The assembler evaluates the term following the `.ASL.` operator as a 16-bit value, and it uses that value as a 16-bit unsigned shift count. The shift count specifies the number of bit positions by which the assembler should arithmetically shift the current 16-bit expression value. In an arithmetic shift to the left the assembler inserts zero bits at the right end of the 16-bit expression value, and it discards bits as they pass to the left out of bit 15, the sign bit. Obviously, any shift count greater than 15 results in a final expression value of zero.

The `.ASR.` operator specifies an arithmetic shift to the right. The assembler evaluates the term following the `.ASR.` operator as a 16-bit value, and it uses that value as a 16-

Table 4.4: Shift and Rotate Operators

Operator	Function
`.ASL.` *(same as `.LSL.`)*	
`.ASR.`	
`.LSL.` *(same as `.ASL.`)*	
`.LSR.`	
`.ROL.`	
`.ROR.`	

bit unsigned shift count that specifies the number of bit positions by which the assembler should arithmetically shift the current 16-bit expression value. In an arithmetic shift to the right the assembler inserts copies of the original sign bit at the left end of the 16-bit expression value, and it discards bits as they pass out of bit zero to the right. As a result, the sign bit is duplicated or extended to the right according to the shift count, and any shift count greater than 14 yields a 16-bit expression that is either −1 or zero depending on the original sign.

The `.LSL.` operator logically shifts the 16-bit expression value to the left. This operator behaves exactly like the `.ASL.` operator. Refer to the documentation about the `.ASL.` operator for more information about the behavior of this operator.

The `.LSR.` operator logically shifts the 16-bit expression value to the right. The assembler interprets the term following the `.LSR.` operator as an unsigned 16-bit shift count. In a logical shift to the right the assembler inserts zero bits at the left end of the 16-bit expression value, and it discards bits as they pass out of bit zero to the right. Any shift count greater than 15 therefore produces an expression value of zero.

The .ROL. operator rotates the 16-bit expression value to the left by the number of bit positions specified by the term following the operator. When a bit rotates out of bit 15 to the left, that same bit rotates into bit zero at the same time. The 16-bit expression value is therefore effectively circular with bit 15 logically adjacent to bit zero. Any rotate count greater than 15 is effectively the same as COUNT.MOD.16 because each 16-bit rotation of the 16-bit expression value returns the expression to its original value. Although shift counts are definitely unsigned, a user can conveniently think of rotate counts as being either signed or unsigned. For example, VALUE.ROL.(-5) yields the same result as a rotate to the right by 5.

The .ROR. operator specifies a rotation to the right by the number of bit positions specified by the term that follows the operator. When a bit rotates out of bit zero to the right, that same bit rotates into bit 15 at the same time, so the 16-bit expression value is effectively circular with bit zero logically adjacent to bit 15. Any rotate count greater than 15 is effectively the same as COUNT.MOD.16 because every 16-bit rotation of the 16-bit expression value leaves the value unchanged. A programmer can think of the rotate count for a .ROR. operator as being either signed or unsigned. For example, VALUE.ROR.(-10) and VALUE.ROL.10 both yield the same result.

For a typical example of a good use of a shift operator, return to the situation in which a person wants to load the address (not the contents) of location VECTOR into the AR and the BR with the AR receiving the most-significant byte of the address of location VECTOR and with the BR receiving the least-significant byte of the address of location VECTOR. A programmer could write "LDAA =VECTOR.LSR.8" as an appropriate instruction to get the most-significant byte of the address of location VECTOR into the AR. The .LSR. operation shifts the most-significant byte of the address of location VECTOR into the least-significant byte of the expression value, thus making the expression value correct for the LDAA instruction with immediate addressing.

A typical example of a good use of a rotate operator occurs when a programmer wants to test a particular bit of the AR. The programmer could write an instruction such as "BITA =1.ROL.5" to test bit 5 of the AR, or the programmer could write an instruction such as "BITA =1.ROL.OVER_RUN" assuming that the label OVER_RUN is defined as a symbolic name for the bit number of a status bit that is set when a data overrun error occurs for a serial I/O port.

4.2.4 Relational Operators

The assembler supports a complete set of signed relational operators for making signed comparisons, and the assembler also supports a complete set of unsigned relational operators for making unsigned comparisons. The relational operators are primarily useful with conditional assembly, but they can, of course, be used in any expression.

A relational operator tells the assembler to make the specified signed or unsigned comparison between the value of the expression on the left of the operator and the value of the term on the right of the operator. The resulting expression value is +1 if the designated

relationship is true, and the resulting expression value is zero if the specified relationship is false. Table 4.5 summarizes the signed relational operators, and Table 4.6 summarizes the unsigned relational operators.

Table 4.5: Signed Relational Operators

Usage	**Meaning**
`<expression>.GT.<term>`	`<expression>` greater than `<term>`
`<expression>.GE.<term>`	`<expression>` greater than or equal to `<term>`
`<expression>.EQ.<term>`	`<expression>` equal to `<term>`
`<expression>.NE.<term>`	`<expression>` not equal to `<term>`
`<expression>.LE.<term>`	`<expression>` less than or equal to `<term>`
`<expression>.LT.<term>`	`<expression>` less than `<term>`

Table 4.6: Unsigned Relational Operators

Usage	**Meaning**
`<expression>.HI.<term>`	`<expression>` higher than `<term>`
`<expression>.HS.<term>`	`<expression>` higher than or same as `<term>`
`<expression>.EQ.<term>`	`<expression>` equal to `<term>`
`<expression>.NE.<term>`	`<expression>` not equal to `<term>`
`<expression>.LS.<term>`	`<expression>` lower than or same as `<term>`
`<expression>.LO.<term>`	`<expression>` lower than `<term>`

For example, the value of the expression `5.GT.3` is 1 since 5 is, in fact, greater than 3, but the value of the expression `5-4.GE.2` is zero since $5 - 4 = 1$, which is not greater than or equal to 2. Also, the value of the expression `(-5).LE.(-3)` is 1 since the relationship is true, and the value of the expression `(-5).LS.(-3)` is also 1 since that relationship is also true. However, while the value of the expression `(-7).LT.5` is 1, the value of the expression `(-7).LO.5` is zero. The `.LO.` operator specifies an unsigned comparison, and -7 equals `$FFF9`, which is definitely not lower than 5 as an unsigned number.

For complete examples that illustrate good uses for the relational operators, please refer to the section that explains conditional assembly.

4.2.5 Selection Operators

The assembler's selection operators return either the expression on the left-hand side or the term on the right-hand side.

The `.MAX.` operator selects the larger of two signed values as the value of an expression, and the `.MIN.` operator selects the smaller of two signed values as the value of an expression. For example, the value of the expression `5.MAX.10` is 10, and the value of the expression `5.MIN.10` is 5. Similarly, the value of the expression `(-5).MAX.(-10)` is -5 since -5 is larger than -10, and the value of the expression `(-5).MIN.(-10)` is -10.

The `.MAXUNS.` operator selects the larger of two unsigned values as the value of an expression, and the `.MINUNS.` operator selects the smaller of two unsigned values as the value of an expression. For example, the value of the expression `5.MAXUNS.10` is 10, and the value of the expression `5.MINUNS.10` is 5. The value of the expression `(-5).MAXUNS.(-10)` is `$FFFB` because -5 equals `$FFFB`, -10 equals `$FFF6`, and `$FFFB` is greater than `$FFF6` when both numbers are considered to be unsigned. Similarly, the value of the expression `(-5).MINUNS.(-10)` is `$FFF6` since `$FFF6` is smaller than `$FFFB` when the numbers are considered to be unsigned. As a final example, the value of the expression `-1.MAXUNS.1` is `$FFFF`.

4.2.6 Unary Operators

The assembler allows one unary operator immediately before any term. Since a parenthesized subexpression is a term, however, you can apply a unary operator to anything by using parentheses. Also, you can apply multiple unary operators to a single term by using parentheses. For example, `.NOT..LOG2.VALUE` is invalid, but `.NOT.(.LOG2.VALUE)` is correct (and also more readable).

As you would expect, the unary operators take precedence over the binary operators. For example, the expression `1+.LOG2.VALUE` is equivalent to `1+(.LOG2.VALUE)` and to `1+(.LOG2.[VALUE])`. The last example is probably the best for clarity.

The unary positive sign (i.e., `+`) emphasizes that the following term is positive, and the unary negative sign (i.e., `-`) negates the following term. Note that the positive sign is not necessary because all terms without a positive or negative sign are positive. For example, $3 * -4 = -12$ and $3 * +4 = 12$.

The `.BITS.` operator returns a count of the one bits in its term. For example, `.BITS.(EXPRESSION)` returns a count of the number of one bits in `EXPRESSION`. A person might use the `.BITS.` operator to compute the number of registers in a register mask.

The logical `.NOT.` operator converts any true value into a false value, and it converts a false value into a true value. The assembler considers any nonzero value to be true, and it considers zero to be false. The `.NOT.` operator produces the value zero for a false output, and it produces the value one for a true output. The logical not operator is intended for use

in logical expressions such as `MPU.EQ.68XX.and..NOT.D2KIT.` The `.NOT.` operator does not produce a one's complement, but the assembler provides a `.COM.` operator for that function.

The `.COM.` operator produces the 16-bit one's complement of a term. Note the difference between the `.NOT.` operator and the `.COM.` operator. The `.NOT.` operator performs a logical operation whereas the `.COM.` operator performs a bitwise inversion.

The `.LOG2.` operator produces the integer portion of the base-2 logarithm of a term. If you express a value in binary, the integer portion of the base-2 logarithm of the value is the bit number of the value's most-significant bit. The `.LOG2.` operator is useful, for example, if you want to substitute a shift for a division or if you want to verify that an expression is a power of 2:

```
        LDAA    =.LOG2.COUNT
1H      LSRB
        DECA
        BGT     1B

        VERIFY  (1.LSL.(.LOG2.COUNT)).EQ.COUNT
```

4.3 Parentheses

The assembler computes the value of an expression from left to right with equal precedence for all operators, so a user doesn't need to memorize any arbitrary precedence relationships. For example, a user doesn't need to know the precedence relationships among the `.AND.`, `.LSR.`, `.GT.`, `.MAX.`, and `.MOD.` operators because all operators have equal precedence.

Although the assembler evaluates expressions from left to right with no operator precedence rules, a programmer can use parentheses to dictate any desired order of evaluation. For example, the expression $5 * (3 + 4)$ evaluates to $5 * 7$ and then to 35 because the parentheses force the assembler to evaluate the parenthesized subexpression as a single term of the expression that contains the parenthesized subexpression.

The assembler allows a programmer to use four different styles of parentheses. In addition to accepting regular parentheses, the assembler also accepts square brackets (`[` and `]`), braces (`{` and `}`), and angular brackets (`<` and `>`). Since the assembler accepts multiple styles of parentheses, we use the generic term "parenthesis" in this section to mean any character that the assembler accepts as a logical parenthesis, not just a "`(`" or "`)`" character. The assembler's multiple styles of parentheses are valuable because they allow a programmer to enhance the readability of a complex expression that contains nested parentheses. For example, consider the following expressions.

```
8*(5+2*(7-4*(9-3*(4-2)+2)-3*(2+1)+7)-6*(4+2)+5)
```

```
8*(5+2*[7-4*{9-3*<4-2>+2}-3*{2+1}+7]-6*[4+2]+5)
```

The two example expressions are functionally identical, but the second expression contains multiple styles of parentheses that improve its readability. While the second expression is still difficult to read correctly, it isn't nearly as bad as the first expression.

The assembler checks each closing parenthesis to verify that it matches the style of the corresponding opening parenthesis, so the multiple styles of parentheses provide some additional error checking that wouldn't be possible with only a single style of parentheses. For example, the assembler reports mismatch errors for expressions like $5 * (4 + 2 * [3 - 4) + 7]$, but a user who typed the same expression with just one style of parentheses would never be aware of the error.

The assembler allows a user to nest parentheses to virtually any level, so programmers don't need to worry about writing expressions that are too complex for the assembler to evaluate. In fact, while an average programmer typically has difficulty understanding more than six levels of nested subexpressions, the assembler easily handles 10,000 levels.

Chapter 5

Assembler Directives

The assembler has an extensive set of assembler directives. Assembler directives, which are sometimes known as pseudo instructions, don't usually correspond to executable machine-language instructions on a one-to-one basis as normal assembly-language instructions do. Instead, an assembler directive might tell the assembler to reserve memory space for a variable, to generate a particular object value for use as a constant, or to define a label to have a specified value. An assembler directive can also tell the assembler to take some desired action or to select some option. For example, the assembler includes directives for generating and formatting a table of contents, adding special references to the concordance, and controlling the format of the program listing. Additionally, the assembler recognizes directives that allow a programmer to control several of the assembler's options.

Table 5.1, which extends over the next several pages, contains a complete list of the assembler's directives, and the remaining part of this chapter contains brief descriptions of the various assembler directives. The assembler directives are presented in alphabetical order, so a reader who wants to refer to the documentation for a particular assembler directive can find the appropriate section quickly and easily. Anything that is shown in the "Parameter" column of Table 5.1 is a descriptive name for a parameter that a programmer can supply in an actual program. The "..." notation in the table indicates that the assembler allows multiple occurrences of the designated field or subfield.

Table 5.1: Summary of Assembler Directives

Directive	Parameter	Remarks
`ASCII`	`"string"`	Generate object byte(s) that contain the ASCII value(s) of the character(s) of the string, and define label as the address of the first data byte of the string.

Continued

Table 5.1 (Continued)

Directive	Parameter	Remarks
ASCIIC	"string"	Generate a data byte that contains an unsigned count of the character(s) in the string, and define label as the address of that data byte. Then generate data byte(s) that contain the ASCII value(s) of the character(s) of the string.
ASCIIZ	"string"	Same as ASCII, but generate a zero (null) byte after the last byte of the string.
ASCT		For relocatable assembly only; start using the location counter for the absolute section.
BHS	destination	Branch if higher or same. This instruction, which is just another name for the BCC instruction, is the unsigned version of BGE.
BLO	destination	Branch if lower. This instruction, which is just another name for the BCS instruction, is the unsigned version of BLT.
BSCT		For relocatable assembly only; start using the location counter for the base section.
BSS	block_size	Block starting at symbol. Skip (i.e., reserve) block_size bytes and name the first one label.
BTEXT		Begin text. Treat the source lines between this statement and the next ETEXT statement as comment lines (except for EJECT, PAGE, SPACE, SPC, SBTTL, and SBTTLE statements).
CHKSUM	target_value	Generate a checksum data byte that can be added to the sum of all object bytes since the previous CHKSUM statement (if any) to yield the specified target_value.
CONC	label1, ...	Create a concordance entry to indicate a reference for each specified label. The CONC statement is used to compensate for invisible references to labels.
CSCT		For relocatable assembly only; start using the location counter for the common section.
DSCT		For relocatable assembly only; start using the location counter for the data section.

Continued

Table 5.1 (Continued)

Directive	Parameter	Remarks
EJECT		Start a new page in the symbolic listing.
END		Terminate the assembly. The assembler ignores anything that appears after the END statement.
ENTRY		Define the entry point of a program. Note that the host environment that executes the program must support this feature in order for this directive to be useful.
EQU	expression	Define the label to equal the value of the expression.
ETEXT		End text. Terminate text mode. The ETEXT statement ends a block of comments.
EXPORT	label1, ...	Export the specified label(s), which must be defined somewhere in this module, to make them available for use as imported labels in other modules that will be linked with this module. The EXPORT statement can be used only in a relocatable assembly. See IMPORT for related information.
FCB	value1, ...	Form constant byte. Generate one data byte for each expression, and name the first data byte label. Each object byte contains the 8-bit signed or unsigned value of the corresponding expression.
FCBS	value1, ...	Similar to the FCB directive except that assembler ensures that the given value is in the 8-bit signed range (i.e., −128 – 127).
FCBU	value1, ...	Similar to the FCB directive except that assembler ensures that the given value is in the 8-bit unsigned range (i.e., 0 – 255).
FCC	"string"	Form constant characters. This statement is the same as the ASCII statement, but the assembler still recognizes the FCC statement for purposes of backward compatibility.

Continued

Table 5.1 (Continued)

Directive	Parameter	Remarks
FDB	value1, ...	Form double byte. Generate two object bytes for each expression. The first object byte of a pair contains the most-significant byte of the corresponding 16-bit signed or unsigned expression value, and the second object byte of a pair contains the least-significant byte of the expression value. Define the label, if any, from the label field to be the address of the first object byte.
GOIF	expression, $target	If the expression is true (i.e., nonzero), go forward to the $target conditional label, thus skipping the assembly of all intervening lines. The GOIF statement provides conditional assembly.
GOIFZ	expression, $target	If the expression ANDed with the CAMASK parameter equals zero, go forward to the $target conditional label, thus skipping the assembly of all intervening lines. The GOIFZ statement can be used with conditional assembly, but it is no longer recommended.
GOTO	$target	Go forward to the $target conditional label, thus skipping the assembly of all intervening lines. The GOTO statement is used with conditional assembly.
GOULZ	expression, $target	Go unless zero. If the expression ANDed with the CAMASK parameter is nonzero, go forward to the $target conditional label, thus skipping the assembly of all intervening lines. The GOULZ statement provides conditional assembly, but it is no longer recommended.
GOUNLS	expression, $target	Go unless. If the expression is false (i.e., zero), go forward to the $target conditional label, thus skipping the assembly of all intervening lines. The GOUNLS statement is used with conditional assembly.

Continued

Table 5.1 (Continued)

Directive	Parameter	Remarks
IMPORT	label1, ...	Import the specified label(s) from other module(s) to make them available for use in this module. The imported labels must be defined as exported labels by the other modules that will be linked with this module. The IMPORT statement can be used only in a relocatable assembly. See EXPORT for related information.
IMPRTB	label1, ...	Similar to the IMPORT directive except that all labels are 8-bit values.
INCLUDE	file_specification	Include the specified source file as part of the program that is being assembled. The specified source file effectively appears in the program immediately after the INCLUDE statement.
MACRO		Define a macro with the specified name and the specified parameters. Note that the macro name appears in the opcode field of the line following the MACRO statement.
MEND		Macro end. Terminate the definition of a macro.
MEXIT		Macro exit. Stop expanding the current macro. The MEXIT statement allows a user to stop expanding a macro before reaching the end of the macro.
MON		Terminate the assembly. The MON statement is equivalent to the more familiar END statement. The assembler recognizes the MON statement to maintain compatibility with Motorola's definition of the assembly language.
NAME	string	The NAME statement is equivalent to the TITLE statement. The TITLE statement is preferred.
OPT	option1, ...	The OPT statement controls the assembler's various options.

Continued

Table 5.1 (Continued)

Directive	Parameter	Remarks
ORG	address	Set the assembler's location counter equal to the value of the address expression, thus specifying the memory location for the next object value.
PAGE		The PAGE statement is equivalent to the EJECT statement. The EJECT statement is preferred.
PRINT	value comment	Print the value of the expression and the text of the comment as a line of the listing and as a message to the user's terminal.
PSCT		For relocatable assembly only; start using the location counter for the program section.
RCB	count, value1, ...	Repeat constant byte. The RCB statement is similar to the FCB statement. The count expression defines a count that tells how many times the assembler should perform the FCB function for the remaining expression values. The result is similar to what would occur if the corresponding FCB statement appeared *count* times.
RCBS	count, value1, ...	Similar to the RCB directive except that assembler ensures that the given value is in the 8-bit signed range (i.e., −128 – 127).
RCBU	count, value1, ...	Similar to the RCB directive except that assembler ensures that the given value is in the 8-bit unsigned range (i.e., 0 – 255).
RDB	count, value1, ...	Repeat double byte. The RDB statement is similar to the FDB statement. The count expression defines a count that tells how many times the assembler should perform the FDB function for the remaining expression values. The result is similar to what would occur if the corresponding FDB statement appeared *count* times.

Continued

Table 5.1 (Continued)

Directive	Parameter	Remarks
RMB	block_size	Reserve memory bytes. Reserve (i.e., skip) block_size bytes, and name the first one label. The RMB statement is equivalent to the BSS statement.
SBTTL	string	Establish string as the current subtitle, and also insert string as an entry in the table of contents. The subtitle string appears on the subtitle line of each page of the symbolic listing, and a user can change the subtitle as often as necessary.
SBTTLE	string	The SBTTLE statement is identical to the SBTTL statement except that the SBTTLE statement also starts a new page in the symbolic listing.
SET	expression	The SET directive is similar to the EQU statement, but the SET statement defines a redefinable label.
SK1		The SK1 statement generates a one-byte executable instruction that skips the next one-byte instruction at execution time.
SK2		The SK2 statement generates a one-byte executable instruction that skips the next two bytes at execution time.
SKIP1		The SKIP1 statement is a more mnemonic name for the older SK1 instruction. The SKIP1 statement generates a one-byte executable instruction that skips the next one-byte instruction at execution time.
SKIP2		The SKIP2 statement is a more mnemonic name for the older SK2 instruction. The SKIP2 statement generates a one-byte executable instruction that skips the next two bytes at execution time.
SPACE	line_count	Skip over line_count blank lines in the symbolic listing, but don't skip any lines at the top of a new page.

Continued

Table 5.1 (Continued)

Directive	Parameter	Remarks
SPC	line_count	The SPC statement is an older name for the SPACE statement. SPACE is now preferred.
TEMPLT	offset	Starts a template section. The given expression in offset is the value of the location counter for this section.
TITLE	string	Establish the string as a title line that appears at the top of each page of the symbolic listing. A user can change the title string as often as desired.
VERIFY	boolean_expression	Verifies that the given expression is true during assembly. This directive helps ensure that assumptions in the code remain true.
XDEF	label1, ...	The XDEF statement is equivalent to the EXPORT statement, but the EXPORT statement is preferred.
XREF	label1, ...	The XREF statement is equivalent to the IMPORT statement, but the IMPORT statement is preferred.

5.1 ASCII

The ASCII statement, which requires a quoted string as an argument, tells the assembler to generate an ASCII data byte for each character of the string.

The string must be enclosed between two identical delimiters, each of which is a single nonwhite character that doesn't appear as a character in the string itself. The following statements illustrate some correct methods for specifying quoted strings.

```
ASCII   "Who is John Galt?"
ASCII   ?Duke Leto is dead.?
ASCII   'Frodo exclaimed, "Watch out, Gandalf!"'
ASCII   A!@#$%^&*()_+-=[]{}:;"'<>,./?\|XYZA
ASCII   ("Go to the Tardis," the Doctor ordered.(
ASCII   "'"
ASCII   /"Live long and prosper."/
ASCII   '"'
ASCII   1991            (Same as "99")
ASCII   RADAR           (Same as "ADA")
```

The following statements show incorrect uses of the `ASCII` directive.

```
ASCII   (Out, out, brief candle!)
ASCII   /The delimiters must match.\
ASCII   'The apostrophe is the user's mistake.'
```

Although the assembler accepts any nonwhite character as a string delimiter, most programmers use quotation marks (`"`) to delimit strings. A programmer who wishes to include a quotation mark as a character in a string ordinarily uses an apostrophe (`'`) as a string delimiter, and a user who wishes to include both quotation marks and apostrophes in a string usually chooses some other special character (such as `!`, `@`, `#`, `$`, `%`, `^`, `&`, `*`, `=`, `;`, `:`, `+`, `/`, or `?`) as a string delimiter.

A programmer should avoid the mistake of trying to use a hyphen as a string delimiter for a string that starts with white space because a hyphen followed by white space at the beginning of the operand field tells the assembler to ignore the rest of the line and to continue the statement with the first nonwhite character of the next line. In this case the assembler does exactly what the user requests, but that might not be what the user really wants. Note that a hyphen followed by white space within a quoted string is perfectly valid as part of the string because the assembler takes all characters within a string literally as characters of the string.

The assembler generates one object data byte for each character of the quoted string, and each object data byte contains the 8-bit ASCII code of the corresponding string character with the parity bit either set or reset according to the parity option that is currently selected. A programmer can specify the parity option (and change it as often as desired) by using the "`OPT PARITY=expression`" assembler directive, which is fully explained elsewhere in this chapter. If a programmer never uses an `OPT` instruction to specify the desired parity, the assembler uses zero parity by default. If zero parity is being used, then, for example, the ASCII code for the character "A" is `$41`. If mark parity is being used, on the other hand, the ASCII code for the character "A" is `$C1`.

If the label field of an `ASCII` statement contains a symbolic label or a local label, the assembler defines the label to equal the memory address of the first byte of the string.

The assembler performs extensive error checking and reports an error if it finds anything wrong with the quoted string that appears as the argument of an `ASCII` statement. For example, the assembler reports an error if a nonwhite character immediately follows the closing string delimiter because the programmer probably tried to use the string delimiter as a character of the string in this case. The assembler also reports an error if it fails to find the closing string delimiter, and it similarly reports an error if the string is empty.

Programmers often use `ASCII` statements to generate character strings that can be used as output strings at execution time, and programmers also use `ASCII` statements to generate strings that can be matched against user inputs at execution time.

5.2 ASCIIC

The `ASCIIC` statement, which is an extension of the `ASCII` statement, produces a counted ASCII string. The assembler's first object output for an `ASCIIC` statement is a data byte that contains an 8-bit unsigned count of the characters, if any, in the string. Then the assembler generates an ASCII data byte for each character of the string just as it would do for an `ASCII` statement.

The following example illustrates an `ASCIIC` statement along with a set of other statements that are essentially equivalent to the given `ASCIIC` statement.

```
RUBICON ASCIIC  "Alea jacta est."

RUBICON FCB     2F-1F
1H      ASCII   "Alea jacta est."
2H      EQU     *
```

Notice from the example that the value of a label that appears in the label field of an `ASCIIC` statement is the memory address of the count byte, not the address of the first character of the string.

Although the assembler reports an error for an empty string with an `ASCII` statement, the assembler allows an empty string (such as `""` or `''`) with an `ASCIIC` statement. An empty string with an `ASCIIC` statement has legitimate uses in some applications, and it causes the assembler to generate a count byte with the value zero to indicate the character count for the empty string.

5.3 ASCIIZ

The `ASCIIZ` statement, which is another extension of the `ASCII` statement, produces a string that is terminated by a zero byte. Like the `ASCII` statement, the `ASCIIZ` statement tells the assembler to generate an ASCII data byte for each character of the string. In addition, the `ASCIIZ` statement tells the assembler to generate a zero (i.e., null) byte immediately following the last byte of the string. A programmer can conveniently use the zero termination byte as a flag that marks the end of the string in memory.

The following example illustrates an `ASCIIZ` statement along with a set of other statements that are functionally equivalent to the given `ASCIIZ`.

```
GAUL    ASCIIZ  "Veni, vidi, vici."

GAUL    ASCII   "Veni, vidi, vici."
        FCB     0
```

As with the ASCIIC statement, the assembler allows an empty string (such as "" or '') with an ASCIIZ statement. An empty string with an ASCIIZ statement causes the assembler to generate just the zero termination byte.

5.4 ASCT

The ASCT statement tells the assembler to start using the location counter that applies to the absolute section of memory. Contrary to what one might first think, the ASCT statement is allowed only in a relocatable assembly, not in an absolute assembly. A programmer can use the absolute section in a relocatable assembly to define I/O ports, timers, and other such items that reside at particular locations in the memory map. For more information concerning the ASCT statement, please refer to the chapter that describes relocatable assemblies.

5.5 BHS

The BHS (branch if higher or same) opcode mnemonic is simply another name for the BCC instruction, which branches if the carry/borrow bit in the condition codes is cleared. Although the BHS instruction is the same as the BCC instruction, the BHS opcode mnemonic conveniently provides a logical extension of the instruction set. When a microprocessor in the 6800 family subtracts one unsigned number (the subtrahend or bottom number) from another unsigned number (the minuend or top number), the carry/borrow bit in the condition codes is cleared if and only if the unsigned value of the minuend or top number is higher than or the same as the unsigned value of the subtrahend or bottom number. The BHS instruction, which branches if the carry/borrow bit is cleared, is therefore useful for comparing unsigned values, and the function that the BHS instruction performs for unsigned numbers is analogous to the function that the BGE instruction performs for signed values.

While the BCC mnemonic is usually appropriate after a shift, a rotation, or an addition, the BHS mnemonic is normally appropriate after a subtraction or a comparison. The availability of the BHS mnemonic allows a programmer to use the mnemonic (BHS or BCC) that most clearly describes the function that the programmer intends to perform.

5.6 BLO

The BLO (branch if lower) opcode mnemonic is simply another name for the BCS instruction, which branches if the carry/borrow bit in the condition codes is set. Although the BLO instruction is the same as the BCS instruction, the BLO opcode mnemonic conveniently provides a logical extension of the instruction set. When a microprocessor in the 6800 family subtracts one unsigned number (the subtrahend or bottom number) from another unsigned number (the minuend or top number), the carry/borrow bit in the condition codes is set if and only if the unsigned value of the minuend or top number is lower than the unsigned value of the subtrahend or bottom number. The BLO instruction, which branches if the

carry/borrow bit is set, is therefore useful for comparing unsigned values, and the function that the BLO instruction performs for unsigned numbers is analogous to the function that the BLT instruction performs for signed values.

While the BCS mnemonic is usually appropriate after a shift, a rotation, or an addition, the BLO mnemonic is normally appropriate after a subtraction or a comparison. The availability of the BLO mnemonic allows a programmer to use the mnemonic (BLO or BCS) that most clearly describes the function that the programmer intends to perform.

5.7 BSCT

The BSCT statement tells the assembler to start using the relocatable location counter that applies to the base section of memory, so the BSCT statement is allowed only in a relocatable assembly, not in an absolute assembly. The primary purpose of the base section is to allow programmers to define relocatable items that can be accessed with direct addressing. The base section's relocatability is therefore severely restricted, and the base section is guaranteed to reside entirely in the memory area that extends from location $0000 through location $00FF. For more information regarding the BSCT statement, please refer to the chapter that describes relocatable assemblies.

5.8 BSS

The BSS (block starting at symbol) directive reserves and names (but does not background) a specified amount of memory for a variable, a table, or some other storage area. The format for a BSS statement is as follows.

```
label   BSS     block_size
```

The expression in the operand field specifies the number of bytes to be reserved. Since the assembler must evaluate this expression during the first pass of the assembly, the expression must not contain any forward references. The assembler effectively reserves the specified amount of memory by adding the value of the expression to the location counter, thus skipping the designated number of bytes in memory.

If a label is present in the label field of a BSS statement, the assembler defines the label to have the original value of the location counter, so the label represents the memory address of the first byte of the reserved area.

The assembler does not generate any object values for the byte(s) that it skips, and it doesn't mark the bytes as being permanently reserved in any way. A programmer can therefore use BSS statements not only to reserve read/write memory for variables, but also to construct templates for data structures and stack frames. For example, the following

instruction sequence defines a template for the stack frame that appears on the stack immediately after an interrupt in the 6800 microprocessor.

```
STACK   EQU     0
        TEMPLT  STACK
CC      BSS     1
BR      BSS     1
AR      BSS     1
XR      BSS     2
PC      BSS     2
```

The statements in the example might initially appear to reserve memory locations starting at location `$0000`, but they are actually being used to define meaningful labels as relative offsets from the first byte of the stack frame, which can be anywhere in memory at execution time. For example, the label AR equals `$0002`, and the value of the microprocessor's AR resides in byte number two of the stack frame where the first byte of the stack frame is byte number zero. A programmer could use a `TSX` instruction after an interrupt to point the XR to the first byte of the stack frame, and then the programmer could use an instruction such as "`STAB AR, X`" to modify the AR value in the stack frame.

5.9 BTEXT

The `BTEXT` (begin text mode) statement marks the beginning of a text block, and it tells the assembler to treat all subsequent lines (with certain exceptions) as comment lines until the next occurrence of an `ETEXT` (end text mode) statement. The assembler recognizes `EJECT` (and `PAGE`), `SPACE` (and `SPC`), `SBTTL`, and `SBTTLE` statements even in text mode because those statements are often just as useful in comment blocks as they are in blocks of code. The assembler also recognizes the `ETEXT` statement in text mode for rather obvious reasons, and it recognizes the `BTEXT` statement in text mode so it can warn a user about a probable missing `ETEXT` statement. Aside from these few statements, the assembler treats all source lines in a text block as comment lines, so a programmer can include any desired documentation in a block of text with no need to type any special comment character at the beginning of each line.

The `BTEXT` statement, like its companion `ETEXT` statement, is suppressed from the symbolic listing, so a text-mode comment block appears in the listing without the distracting clutter of the `BTEXT` and `ETEXT` statements that bracket it.

If a programmer uses a `BTEXT` statement and forgets to include the matching `ETEXT` statement, the assembler eventually reports at least one error. In the worst case, the assembler processes the rest of the program (including the `END` statement) as a text block and then reports that the `END` statement is missing. More commonly, the assembler reports a redundant `BTEXT` statement the next time it finds a `BTEXT` directive, thus alerting the user to the fact that an `ETEXT` statement was probably omitted somewhere.

5.10 CHKSUM

The CHKSUM statement generates a checksum data byte with a unique 8-bit value that can be added to the 8-bit sum of all object bytes since the previous CHKSUM statement (if any) to yield a specified 8-bit checksum target value. The CHKSUM statement is therefore useful to a programmer who wants to write a program that verifies the integrity of a ROM at execution time. The format for the CHKSUM statement is as follows.

```
label   CHKSUM  target_value
```

The expression in the CHKSUM statement's operand field specifies the desired target value for the checksum. Most programmers use −1 as the target value, but the assembler allows any 8-bit value. Although the assembler accepts any 8-bit target value, the value zero is specifically not recommended because a common kind of hardware failure can cause every location in an entire ROM to read as a zero byte, thus passing a checksum verification if the target checksum is zero.

A programmer typically uses a CHKSUM statement to generate a checksum byte as the last byte of a ROM or as the last byte of a group of ROMs. The programmer who wants to use the checksum byte to validate the ROM writes a self-test routine that computes the 8-bit sum of all of the bytes of the ROM, including the checksum byte. The self-test routine simply ignores both overflows and carries that occur during the dynamic computation of the checksum value because only the final 8-bit result is meaningful. The self-test routine compares the computed checksum value to the known target value, and the routine reports an error in the ROM if the computed checksum value doesn't match the target value.

The following example code segment is a self-test routine that checksums a ROM. The ROM starts at location ROM and contains LENGTH bytes, and the target value for the checksum was specified as TARGET.

```
        CLRA
        LDX     =ROM

1H      ADDA    0, X
        INX
        CPX     =ROM+LENGTH
        BNE     1B

        CMPA    =TARGET
        BNE     ERROR
```

In some cases the checksum byte can't be the last byte of a ROM because a system's hardware might require special values such as interrupt vectors at the end of a ROM. In

such a case a clever programmer can place the checksum byte elsewhere and still include the interrupt vectors in the checksum. The programmer can simply use an ORG statement to put the interrupt vectors at their special locations and then use another ORG statement followed by a CHKSUM statement to generate the checksum byte at its location. Thus the checksum byte can cover an entire ROM without being at the last location in the ROM.

The assembler also allows the CHKSUM statement in a relocatable assembly, and the linker properly generates the specified checksum value to cover the object values generated in all program sections together. If you want the checksum to cover contributions from all modules, you must specify the module containing the CHKSUM statement as the last module in the link.

5.11 CONC

The CONC statement generates a concordance entry to indicate a reference for each label that appears in the CONC statement's operand field. The purpose of the CONC statement is to generate concordance entries for labels that are accessed invisibly by a nearby statement. The format for the CONC statement is as follows.

```
        CONC    <expression-1>, <expression-2>, ..., <expression-n>
```

The CONC statement requires at least one expression, and it accepts any number of expressions beyond the first one. In actual practice, each expression is normally just a single label.

An example might be beneficial. Suppose that a program maintains several variables: COUNT, ID, LINK, MACRO, SEGMENT, SIZE, VALUE, and WEIGHT. Suppose furthermore that by design these variables reside together in a sequential array and that an initialization routine in the program is supposed to clear all of these variables. The initialization routine could clear each variable individually by name, or it could simply use a loop to clear the entire array of variables. The following example code segment illustrates the use of a loop to clear the array of variables.

```
        LDX     =ARRAY
1H      CLR     0, X
        CONC    COUNT, ID, LINK, MACRO, SEGMENT, SIZE, VALUE, WEIGHT
        INX
        CPX     =ARRAY+LENGTH
        BNE     1B
```

The CLR instruction in the example clears each variable in turn, but it doesn't use any of the variable names. The CLR instruction therefore makes invisible references to

the variables, and these invisible references don't show up in the concordance listing, thus severely handicapping anyone who later tries to maintain or modify the program. The `CONC` statement eliminates the deficiency of the invisible references by generating an artificial concordance entry for each variable name that appears in the `CONC` statement's operand field.

5.12 CSCT

The `CSCT` statement tells the assembler to start using the relocatable location counter that applies to the common section of memory, so the `CSCT` statement is allowed only in a relocatable assembly, not in an absolute assembly. The common section is a relocatable area of uninitialized memory that is shared among modules, usually for common variables or for scratch memory. Each module's common section overlays the same area of memory, so the common section is analogous to FORTRAN's blank common. For additional information concerning the `CSCT` statement, please refer to the chapter that describes relocatable assemblies.

5.13 DSCT

The `DSCT` statement tells the assembler to start using the relocatable location counter that applies to the data section of memory, so the `DSCT` statement is allowed only in a relocatable assembly, not in an absolute assembly. The data section can be relocated to any memory address at link time, so accesses to items in the data section require extended addressing as opposed to direct addressing. The data section normally resides in read/write memory, and it is normally used for variables that can't be put into the base section. For more information regarding the `DSCT` statement, please refer to the chapter that describes relocatable assemblies.

5.14 EJECT

The `EJECT` statement tells the assembler to advance the pass-2 symbolic listing (if one is being produced) to the top of a new page. If the listing already happens to be at the top of a page when the assembler encounters an `EJECT` statement, the assembler doesn't advance to the top of another page, so the assembler never wastes paper by putting unnecessary blank pages into the middle of a program listing. The assembler doesn't print the `EJECT` statement itself because that would clutter the listing. The `EJECT` statement doesn't have an operand field, of course, and it must not have a label in its label field.

A user normally puts an `EJECT` statement at the beginning of every logical section of his or her program to improve readability.

5.15 END

The END statement terminates the assembly, so it must be the last statement in every program. The assembler reports an error if the END statement is missing, and it simply ignores any lines that occur after the END statement in the source program. The END statement doesn't have an operand field, and its label field must be empty. If an END statement appears in an included file (see INCLUDE), it terminates the entire assembly just as it would if it appeared in the main source file.

5.16 ENTRY

The ENTRY pseudo instruction allows the user to specify a starting point in the user program. If a program contains an ENTRY statement, a debugging program (e.g., MUDBUG) can set the program counter to the program's entry point when you download the program. The following example illustrates the ENTRY statement.

```
START   ENTRY                    Program execution begins here.
                                The START label is optional.   If a
                                label is present, it gets the value
                                of the location counter.
```

An ENTRY statement does not generate any object code itself. It merely indicates where execution of the program should begin. An ENTRY statement is not appropriate in a module such as a subroutine module that does not run as an independent program.

5.17 EQU

The EQU statement equates the label in the label field to the value of the expression that appears in the operand field. Since the assembler records values for labels during pass 1, no forward references are allowed in the operand field of an EQU statement.

5.18 EQUREG (6809)

The EQUREG statement is analogous to the EQU statement, but the EQUREG statement takes a register list instead of an expression as its operand.

5.19 ETEXT

The ETEXT (end text mode) statement terminates a text block that was started by a BTEXT (begin text mode) statement. Like the BTEXT statement, the ETEXT statement is not

printed in the symbolic listing, so a text-mode comment block appears in the listing without the distracting clutter of the BTEXT and ETEXT statements that bracket it.

If the assembler encounters an ETEXT statement without first processing a matching BTEXT statement to start a text block, the assembler reports an error. Please refer to the description of the BTEXT statement for related information.

5.20 EXPORT

The EXPORT statement tells the assembler to export the specified label(s) to make them available for use as imported labels in other program(s) that will eventually be linked with the program that is being assembled. The EXPORT statement is valid only in a relocatable assembly, and the label(s) that it specifies must be defined somewhere in the program that is being assembled. The format for the EXPORT statement is as follows.

```
        EXPORT  label_1, label_2, ..., label_n
```

The EXPORT statement requires at least one label as an operand, and it accepts any number of labels beyond the first one. The EXPORT statement's label field must be empty.

For more information regarding the EXPORT statement, please refer to the chapter that describes relocatable assemblies, and also refer to the description of the related IMPORT statement.

5.21 FCB

The FCB (form constant byte) statement tells the assembler to evaluate the operand expression and to generate an object data byte that contains the 8-bit signed or unsigned value of the expression. The FCB statement is therefore useful for creating a constant value that can be accessed by a program at execution time. The assembler allows both signed and unsigned values for FCB expressions, so the assembler accepts an expression if its value fits properly into 8 bits either as a signed value or as an unsigned value. The format for an FCB statement is as follows.

```
label   FCB     value_1, value_2, ..., value_n
```

The FCB statement requires at least one operand expression, and it allows any number of operand expressions after the first one. An FCB statement with multiple operand expressions is effectively a shorthand notation for a series of individual FCB statements, so a single FCB statement with multiple operand expressions is equivalent to a series of individual FCB statements with the same set of operand expressions.

The assembler advances the location counter after it processes each expression of an FCB statement, so the location-counter symbol (*) in an FCB statement always refers to the memory address of the data byte that is being generated by the expression that contains the location-counter symbol.

If a label appears in the label field of an FCB statement, the assembler defines that label's value to be the memory address of the first data byte that is generated by the FCB statement.

5.22 FCBS

The FCBS directive is like FCB, but the FCBS statement specifies that the assembler should interpret and validate the expression value as a signed number.

5.23 FCBU

The FCBU directive is like FCB, but the FCBU statement specifies that the assembler should interpret and validate the expression value as an unsigned number.

5.24 FCC

The FCC (form constant characters) opcode mnemonic is simply another name for the ASCII opcode mnemonic. The assembler recognizes the FCC mnemonic to provide compatibility with other assemblers, but most users prefer the more descriptive ASCII opcode mnemonic.

The FCC statement, which requires a quoted string as an argument, tells the assembler to generate an ASCII data byte for each character of the string. Please refer to the description of the ASCII statement for more details.

5.25 FDB

The FDB (form double byte) assembler directive is similar to the FCB (form constant byte) assembler directive, but an FDB statement generates a 16-bit constant that occupies two consecutive object bytes while an FCB statement generates an 8-bit constant that occupies a single object byte. The FDB statement is useful for creating a 16-bit constant value that can be accessed by a program at execution time.

The FDB statement tells the assembler to evaluate the operand expression and to generate two object data bytes that contain the 16-bit signed or unsigned value of the expression. The most-significant half of the 16-bit expression value for an FDB statement occupies the first object byte, and the least-significant half of the 16-bit expression value occupies the second object byte. Note that the assembler allows both signed and unsigned values for FDB expressions. The format for an FDB statement is as follows.

```
label   FDB     value_1, value_2, ..., value_n
```

The FDB statement requires at least one operand expression, and it allows any number of operand expressions after the first one. An FDB statement with multiple operand expressions is effectively a shorthand notation for a series of individual FDB statements, so a single FDB statement with multiple operand expressions is equivalent to a series of individual FDB statements with the same set of operand expressions.

The assembler advances the location counter after it processes each expression in an FDB statement's operand field, so the location-counter symbol (*) in an FDB statement always refers to the memory address of the first object data byte that is being generated by the expression that contains the location-counter symbol.

If a label appears in the label field of an FDB statement, the assembler defines that label's value to be the memory address of the first data byte that is generated by the FDB statement.

5.26 GOIF

The GOIF statement allows a user to perform conditional assembly by skipping a conditional section of source code or including the conditional section of source code as part of the program during the assembly process. The format for the GOIF statement is as follows.

```
        GOIF    conditional_expression, $target
```

If the conditional expression is false (i.e., equal to zero), the assembler simply continues the assembly with the next line, and the GOIF statement essentially has no effect. If, on the other hand, the conditional expression is true (i.e., nonzero), the assembler goes forward to the $target conditional label, thus skipping the assembly of all intervening lines. The $target value in the GOIF statement is a 16-bit hexadecimal number that matches the value of a $target conditional label that appears in the label field of a subsequent target line. The assembler requires the use of hexadecimal numbers for conditional labels much the same as FORTRAN requires the use of decimal numbers for statement labels.

Obviously, the assembler must evaluate the GOIF conditional expression during the first pass of the assembly, so forward references are not allowed in a GOIF conditional expression. For more information regarding the GOIF statement, please refer to the chapter that describes conditional assembly.

5.27 GOIFZ

Like the GOIF statement, the GOIFZ (go if zero) statement allows a user to perform conditional assembly by skipping a conditional section of source code or including the conditional section of source code as part of the program during the assembly process. The

GOIFZ statement was the primary vehicle for performing conditional assembly before the GOIF statement was added to the assembler's repertoire, but the GOIFZ statement is no longer recommended for use. Anything that can be done with the GOIFZ statement can be done more clearly with the GOIF statement, and the assembler still recognizes the old GOIFZ statement only for purposes of backward compatibility. The format of the GOIFZ statement is as follows.

```
GOIFZ   conditional_expression, $target
```

If the conditional expression ANDed with the assembler's CAMASK parameter is nonzero, the assembler simply continues the assembly with the next line, and the GOIFZ statement essentially has no effect. If, on the other hand, the conditional expression ANDed with the CAMASK parameter is zero, the assembler goes forward to the $target conditional label, thus skipping the assembly of all intervening lines. The CAMASK parameter is a 16-bit value that a programmer can dynamically specify as a run-time parameter when the programmer invokes the assembler. The CAMASK parameter is discussed in the section that explains the assembler's run-time options.

The assembler must evaluate the conditional expression in a GOIFZ statement during the first pass of the assembly, so forward references are not allowed in a GOIFZ conditional expression. For more information regarding the GOIFZ statement, please refer to the chapter that describes conditional assembly.

5.28 GOTO

Like the GOIF statement, the GOTO statement is used to skip a section of source code during the assembly process. Unlike the GOIF statement, however, the GOTO statement unconditionally skips a section of source code. The format of the GOTO statement is as follows.

```
GOTO    $target
```

The $target value in the GOTO statement's operand field is a 16-bit hexadecimal number that matches the value of a $target conditional label that appears in the label field of a subsequent target line. The assembler uses hexadecimal numbers for conditional labels much the same as FORTRAN uses decimal numbers for statement labels.

The unconditional GOTO statement often appears in conjunction with the conditional GOIF statement to perform an "if-then-else" conditional function for a user. A programmer frequently wants to assemble one of two alternative sections of a program. In this case the programmer can use a GOIF statement to skip or include the first alternative section, and

the programmer can use a GOTO statement at the end of the first alternative section of the program to skip the second alternative section when the GOIF statement selects the first alternative section.

The GOTO statement is also useful when a programmer wants to omit a section of a program without deleting that section from the program's source file. The programmer simply uses a GOTO statement to skip the section in question at assembly time, but the skipped section can be recovered easily since it is still present in the program's source file.

For more information regarding the GOTO statement, please refer to the chapter that describes conditional assembly.

5.29 GOULZ

The GOULZ (go unless zero) statement is the logical complement of the GOIFZ (go if zero) statement. Like the GOIFZ statement, the GOULZ statement allows a user to perform conditional assembly by skipping a conditional section of a source program or including the conditional section of the source program as part of the program during the assembly process. Also like the GOIFZ statement, the GOULZ statement is no longer recommended for use, and the assembler still recognizes the GOULZ statement only for purposes of maintaining compatibility with old programs. Anything that can be done with the GOULZ statement can be done more clearly with the GOIF statement, so programmers should avoid using the GOULZ statement in any new programs. The format for the GOULZ statement is as follows.

```
GOULZ   conditional_expression, $target
```

If the conditional expression ANDed with the assembler's CAMASK parameter is zero, the assembler simply continues the assembly with the next line, and the GOULZ statement essentially has no effect. If, on the other hand, the conditional expression ANDed with the CAMASK parameter is not zero, the assembler goes forward to the $target conditional label, thus skipping the assembly of all intervening lines. The CAMASK parameter is a 16-bit value that a programmer can dynamically specify as a run-time parameter when the programmer invokes the assembler. The CAMASK parameter is discussed in the section that explains the assembler's run-time options.

The assembler must evaluate the conditional expression in a GOULZ statement during the first pass of the assembly, so forward references are not allowed in a GOULZ conditional expression. For more information regarding the GOULZ statement, please refer to the chapter that describes conditional assembly.

5.30 GOUNLS

The GOUNLS (go unless) statement, which allows a user to perform conditional assembly, is the logical complement of the GOIF statement. The format for the GOUNLS statement is as follows.

```
GOUNLS  conditional_expression, $target
```

If the conditional expression is true (i.e., not equal to zero), the assembler simply continues the assembly with the next line, and the GOUNLS statement essentially has no effect. If, however, the conditional expression is false (i.e., zero), the assembler goes forward to the $target conditional label, thus skipping the assembly of all intervening lines. The assembler must evaluate the GOUNLS conditional expression during the first pass of the assembly, so forward references are not allowed in a GOUNLS conditional expression.

The GOUNLS statement isn't really necessary because a programmer can always use a GOIF statement to do anything that a GOUNLS statement can do. The assembler supports the GOUNLS statement, however, because the reverse logic of the GOUNLS statement can be more clear than the GOIF statement's positive logic in some situations.

For more information regarding the GOUNLS statement, please refer to the chapter that describes conditional assembly.

5.31 IMPORT

The IMPORT statement tells the assembler to import the specified label(s) from other program(s) to make them available for use in the program that is being assembled. The IMPORT statement is valid only in a relocatable assembly, and the label(s) that it specifies must be defined as exported labels in other module(s) that will eventually be linked with the program that is being assembled. The format for the IMPORT statement is as follows.

```
IMPORT  label_1, label_2, ..., label_n
```

The IMPORT statement requires at least one label as an operand, and it accepts any number of labels beyond the first one. The IMPORT statement's label field must be empty.

The .B size qualifier can follow a label in the import list for the IMPORT statement. The following example illustrates the syntax:

```
IMPORT LABEL1.B, LABEL2, LABEL3.B, LABEL4, LABEL5, LABEL6.B
```

The .B size qualifier tells the assembler that the imported label is a byte-sized label that is not relocatable beyond the range of an 8-bit address, so the assembler can choose to use direct addressing for an access to the label even though the label's actual value is unknown at assembly time.

For more information regarding the IMPORT statement, please refer to the chapter that describes relocatable assemblies, and also refer to the description of the related EXPORT statement.

5.32 IMPRTB

The IMPRTB statement is just like the IMPORT statement, but all of the labels in the import list for an IMPRTB statement are byte-sized imports (with or without the .B size qualifier).

5.33 INCLUDE

The INCLUDE statement requires a standard VMS file specification in its operand field, and it tells the assembler to include the specified source file as part of the program that the assembler is assembling. The specified source file effectively appears in the program immediately after the INCLUDE statement. The assembler treats source lines from an INCLUDE file exactly the same as lines from the main source file.

The INCLUDE statement itself does not appear in the listing, so users may find it a good practice to make the first line of an included file a comment that identifies the included file.

The assembler permits nesting of INCLUDE statements to any depth, but it does not support recursion at any level. No included file can include itself, nor can any nested include file contain an INCLUDE statement that refers to a file higher up in the nested hierarchy.

Invalid file names or nonexistent files, file read system errors, or the unavailability of virtual memory to process file control blocks, are types of error conditions that the include processor reports. See Chapter 10 for more details on these errors.

5.34 END

An END statement in an included file will terminate the assembly, and the assembler will ignore any subsequent source lines in the main source file and in the possibly nested structure of include files.

5.35 MACRO

The MACRO instruction introduces a macro definition. Everything from the MACRO pseudo-op to the next MEND pseudo-op is interpreted as part of the macro definition. Since the MACRO instruction is not associated with the location counter as it does not generate object code, the MACRO instruction must not be labeled. Refer to Chapter 8 for information on the use of macros.

5.36 MEND

The MEND instruction ends a macro definition. When the assembler encounters a MACRO pseudo-op, it processes everything until the next MEND instruction as lines of the macro

definition. The MEND statement must not be labeled, and it is only valid in a macro definition. If a MEND instruction is labeled, the macro processor will not terminate the definition. If a MEND occurs outside of a macro definition, the assembler will report a non-fatal error. Refer to Chapter 8 for information on the use of macros.

5.37 MEXIT

The MEXIT (macro exit) instruction tells the macro expansion processor to exit a macro definition. When a macro call occurs, program lines are generated by expansion of the macro definition model lines. (Normally, program lines are retrieved from the source file.) Expansion ends when a MEXIT or a MEND instruction is encountered. Unlike the MEND instruction, MEXIT instructions do not end a macro definition, only the expansion of that definition. MEXIT is sometimes used in conjunction with conditional assembly to prevent expansion of lines that will be skipped anyway. The MEXIT instruction must not be labeled, and it is only valid in a macro definition. If a MEXIT instruction is labeled, or if a MEXIT occurs outside of a macro definition, the assembler will report a non-fatal error. Refer to Chapter 8 for information on the use of macros.

5.38 MON

The MON instruction is equivalent to the END statement, and the MON statement is included only for compatibility with the Motorola definition of the assembly language. Please refer to the description of the END statement for more information.

5.39 NAME

The NAME statement is equivalent to the TITLE statement, and the NAME statement is included only for compatibility with the Motorola definition of the assembly language. Please refer to the description of the TITLE statement for more information.

5.40 OPT

The OPT pseudo instruction is used to set or reset various assembly options during pass 1 and/or pass 2 of the assembly. Some of the options are also selectable during run-time processing at the user-interface level as described in Chapter 6. An OPT command in a source program, however, will override any run-time option selected by the user if the options are in conflict. All of the options are initially set to their most commonly used values by the assembler, and so most users don't need to worry about setting options during assembly of their programs.

When an option is needed in a program, and the program will always need this option (and the option is not one of the default values), it is best to include the option in the

program using the OPT pseudo instruction. If on the other hand, this option is a one-time selection, it is best to use the run-time option, if it is available. For example, if a program's output listing is to always start at page number 500, by including the assembly-time statement "OPT PAGENUM=500" in the source program, the user doesn't have to remember to specify the starting page number each time the program is assembled. If, on the other hand, the program is to be assembled with a starting page number of 500 for just one special run, it is better to specify the initial page number with the "Page=n" run-time option (so the user needn't insert a nonstandard instruction into the source code, which he or she just might forget to remove after the assembly).

Some of the options are either set or reset (i.e., selected or not), some have levels associated with them, and still others require specific value assignments. A sample input for the option selections for outputting in base 10, an initial page number of 5, and a label length limit of 8 may be specified as:

```
OPT     DB10, PAGENUM=5, LBLLIM=8
```

The options can be specified in any order, and options can be specified in any combination of uppercase and lowercase characters. However, it is good programming practice to use uppercase for the instructions, and to use lowercase only in the comment field. Except for the TABS option, several options may be specified in a single OPT statement by simply separating one option field from the next by a comma (or a comma and white space). Option names that are under six characters must appear exactly as listed in Table 5.2. Option names given in Table 5.2 that are longer than six characters (these are generally spelled out for the purposes of readability) can be specified in any fashion as long as the first six characters of the name exactly match that given in the table. If the assembler finds an invalid option, it reports an invalid option error (??IO) and continues to process the option list.

The remainder of this section describes the various OPT assembly options, and although it is not specifically mentioned in the following paragraphs, each of the options selected must be in the operand field, and an OPT pseudo opcode must be specified in the opcode field of the same statement. An alphabetized table of OPT options is also provided as a quick and convenient reference.

Table 5.2: Summary of Assembler OPT Statements

Option	**Remarks**
ABS, REL	Select between absolute or relocatable object code generation.
CALIST, CANOLIST	Increment or decrement respectively the conditional assembly list level.

Continued

Table 5.2 (Continued)

Option	Remarks
CMTCOL=n	Any statement that begins with its first significant character more than n columns from the beginning is considered to be a comment line.
DB8, DB10, DB16	Select the display base for the object code in the assembly listing as octal, decimal or hexadecimal, respectively.
DEBUG	Include the symbol table in the object file for debugging purposes.
DIRECT, NODIRECT	Enable or disable the prerogative of the assembler to generate object code with the direct addressing mode.
EOERR, NOEOERR	Report or do not report macro expansion overflow errors to the assembly listing.
FORMAT, NOFORMAT	Select or deselect automatic source line formatting. This option is provided for backward compatibility only.
IGNORE, NOIGNORE	Suppress or list, respectively, source lines that start with a period in column one.
INCLIST, INCNOLIST	Include or suppress, respectively, the INCLUDE statements and files from the listing.
LBLLIMIT=n	Set the number of significant characters in a symbolic label equal to n.
LINENUMBER=n	Set the current line number of the assembly listing to n. Subsequent source lines are then incremented from this value.
LIST, NOLIST	Increment or decrement respectively, the listing level in the assembly listing.
LLRANGE=n	Set the local label range equal to n. Any references to a local label outside of this range of line numbers generates an assembly error.
LONG, SHORT	List or suppress respectively, multiple object lines in the assembly listing. Assembler directives such as FCC or FDB can produce multiple object lines.
LPAGE=n	Set the length of the logical page to n. Minus the two standard header lines, LPAGE is the number of lines of user output that will appear per physical page.
MPERR, NOMPERR	Report or do not report any macro missing parameter errors to the assembly listing.

Continued

Table 5.2 (Continued)

Option	Remarks
MPU=xxxx	Assemble the subsequent lines for the specified microprocessor, xxxx. Valid values for xxxx are 6800, 6801, 6802, 6803, 6805, 6808, 6809, 6811, 6301, 6502, and 65150.
MSHOW, NOMSHOW	Include or suppress respectively, source lines that result from macro expansion.
PAGENUMBER=n	Begin the pagination in the assembly listing with the number n.
PARITY=n	Set the parity bit of any ASCII characters generated in the assembly to n. When n equals 1 mark parity is selected. When n equals 0 space or zero parity is selected.
RV8, NORV8	In relocatable mode, give a warning if an 8-bit expression may exceed the 8-bit range during link time if NORV8 is selected. Otherwise, do not give such warnings.
TABS=n1,n2,...,nk	Set the tab stop in the columns denoted by n1 through nk. A "TABS OPT" statement must appear in its own OPT statement.
TMERR, NOTMERR	Report or do not report respectively, the macro too-many-parameters errors to the assembly listing.

5.40.1 OPT ABS/REL

The ABS and REL switches to this OPT statement select absolute or relocatable assembly respectively. Absolute assembly is the default mode of object code generation. When the user selects relocatable assembly, a number of assembler directives become active to enable part of the assembly work to be done by the linker. The user is referred to Chapter 9 on relocatable assembly and to the coverage in this chapter of such directives as IMPORT, EXPORT, ASCT, BSCT, CSCT, DSCT, and PSCT for more information.

The analogous run-time options for determining mode of object code generation are "ABS" and "REL".

5.40.2 OPT CALIST/CANOLIST

By default the assembler lists all statements that are associated with conditional assembly, but the user can change between the list mode and the no-list mode for conditional-assembly statements as often as desired by selecting the CALIST or CANOLIST option, respectively. There are levels associated with the list mode for conditional-assembly statements. The CANOLIST option decrements the list level, and the CALIST option increments the list level. The list level is initially zero, and a negative list level will result in the suppression of conditional-assembly statements. Therefore, the user can bracket a program segment

with the CANOLIST/CALIST pair with no net change to the current list level for conditional-assembly statements, i.e., CANOLIST and CALIST options at earlier points in the program remain in effect after a subordinate CANOLIST/CALIST pair have been processed.

The CANOLIST option suppresses GOIF, GOUNLS, GOIFZ, GOULZ and GOTO statements, statements with conditional-assembly labels, and skipped statements (i.e., statements that won't be assembled because of a conditional-assembly action) from the output listing. (See Chapter 7 for a discussion of conditional-assembly statements, and see Chapter 6 for a description of the corresponding run-time list-conditional-assembly option.)

The default for the CANOLIST/CALIST option is CALIST (i.e., zero).

5.40.3 OPT CMTCOL

The CMTCOL=n option allows the user to specify the number of blank columns that may occur between the end of the label field and the occurrence of a nonblank character before the line is considered to be a comment statement. If there is no label in the label field, the column counting starts with column one.

An invalid-option error (??IO) is reported unless $1 < \mathtt{n} \leq 80$. CMTCOL is set to a default value of 10 at the beginning of each pass, but it may be changed as often as desired. (Refer to the example in Chapter 2 under the discussion of the label field.) This feature alleviates the user from having to start comment lines with an asterisk, provided that there are at least n blank columns preceding the first nonblank character of the comments.

The default value for the CMTCOL parameter is 10.

5.40.4 OPT DB8/DB10/DB16

The user may specify either the DB8, DB10, or the DB16 option, and the generated pass-2 values will then be output in base 8 (octal), base 10 (decimal), or base 16 (hexadecimal) character format, respectively. If none of the options are specified, the assembler will automatically output the values in hexadecimal. The display base may be changed as often as desired in the program.

Some programmers who have trouble figuring out displacement values in hexadecimal may find the decimal base selection as an aid in debugging, but it is not recommended since most programmers are expected to become experts in binary, octal, and hexadecimal arithmetic. The value of the location counter is always printed in hexadecimal regardless of the display base, but object values are printed according to the display base selected.

The default for the display-base option is base 16, i.e., hexadecimal.

5.40.5 OPT DEBUG

By using this option, the user can include a symbol table in the generated object file. Each entry in the symbol table consists of an ASCII-string label and its corresponding

numeric value. Thus, a debugging tool (e.g., MUDBUG) can help the programmer by providing labels in place of "magic" numbers. This debugging option is available in both absolute and relocatable modes.

5.40.6 OPT DIRECT/NODIRECT

Under certain conditions the user may need to restrict the assembler from generating object code that uses the direct addressing mode. In these situations, the `OPT` switch, `NODIRECT`, can be selected. By default, the assembler generates object code with direct addressing whenever expressions in the operand field of instructions that permit this mode can be evaluated in pass 1.

5.40.7 OPT EOERR/NOEOERR

By default, the assembler reports an error if the expanded macro line overflows the source-line buffer (i.e., exceeds 126 characters). This error occurs when the length of the substitution string exceeds the length of the parameter name and the macro model line did not have enough unused spaces at the end to absorb the difference. For example, if `\1` is replaced by `ALPHA + BRAVO - CHARLIE/DELTA`, the resulting macro expanded line is going to contain more characters than the model line does.

Since the programmer may determine that nothing of value was lost in the expanded line (only the comment field was truncated), the assembler allows the user to suppress reporting of the expansion overflow error. The user should not routinely suppress the error; it is best to suppress the error to get a "clean" listing only after ensuring that nothing important is lost.

The user can change between the report expansion overflow mode and the suppress expansion overflow error mode as often as desired by selecting the `EOERR` or `NOEOERR` option, respectively. There are also run-time `EO` and `NEO` options, and the `OPT` source-code options override the run-time options.

The default for the `EOERR/NOEOERR` option is `EOERR`.

5.40.8 OPT FORMAT/NOFORMAT

The `FORMAT` option directs the assembler to automatically format the user's assembly-language listing into neatly aligned columns for the label, opcode, operand, and comment fields. The `NOFORMAT` option, on the other hand, leaves the formatting to the user through the use of space and tab characters. The no-format mode is assumed by default at the beginning of each pass of the assembly, but the user can change between no-format and format modes as often as may be desired. Format mode is of little use in view of the fact that users can conveniently use horizontal tabs, but the format mode is retained as

an optional feature of the assembler to provide compatibility with earlier versions of the assembler that existed before the advent of modern disk-based text editors.

The default for this option is `NOFORMAT`.

5.40.9 OPT IGNORE/NOIGNORE

By default, the assembler ignores or suppresses from the assembly listing any source line that starts with a period. Often these dot lines are the commands for a text formatter that the user will employ to improve the quality and readability of the listing. However, when a program is still under development, the user may not want to squander the computer resources to preprocess the source with the formatter when the listing is only a preliminary version. The `IGNORE` option permits a reasonably clear listing since it causes the assembler to omit these temporarily extraneous source lines from the listing. The `NOIGNORE` option allows the user to display the dot lines in the assembly listing. Run-time variants of this option are described in Chapter 6.

5.40.10 OPT INCLIST/INCNOLIST

"`OPT INCLIST`" tells the assembler to list `INCLUDE` statements in the symbolic listing, and "`OPT INCNOLIST`" tells the assembler to suppress `INCLUDE` statements from the symbolic listing. The assembler now lists `INCLUDE` statements by default unless user includes an "`OPT INCNOLIST`" statement. You can nest the "`OPT INCLIST`" and "`OPT INCNOLIST` statements much as you can nest `BEGIN-END` statements in Algol, and a nested `INCLIST/INCNOLIST` pair preserves the current listing option.

5.40.11 OPT LBLLIMIT

The `LBLLIMIT=n` option tells the assembler to consider only the first `n` characters of a label to be significant and to ignore all characters after the `n`th one. Labels can be as long as the user wants, up to and including a limit of 80 characters, but only the first `n` characters of a label are significant. The default value for the `LBLLIMIT` option is 80, but the user can reduce its value to conserve space in the symbol table and to make the assembly faster. Note that care must be taken not to change the label limit between the occurrence of a label in the label field and its use in an operand field since the labels will not be the same if there is even a single character difference in the significant lengths. An invalid-option error is reported unless $\texttt{n} > 0$.

The default label limit is 80.

5.40.12 OPT LINENUMBER

The `LINENUMBER=n` option tells the assembler to set the line number of this line equal to `n` and to restart the line-numbering sequence from this point. The assembler accepts any

16-bit value for the line number, and it interprets the 16-bit value as an unsigned number. The default initial line number is one, but the user can start a new line-number sequence as many times as desired in the program. The initial line number may also be selected with the `LIN` (LINe number) run-time option (see Chapter 6).

The default value for the `LINENUM` option is 1.

5.40.13 OPT LIST/NOLIST

The `LIST` option tells the assembler to produce a symbolic listing of the program (provided, of course, that the user has specified an output device or file for the listing), and the `NOLIST` option suppresses any symbolic assembly listing. The `LIST` and `NOLIST` options may be used in various portions of the program to generate and suppress lines from the symbolic output listing (e.g., suppress previously debugged routines from being output).

The `LIST/NOLIST` option uses listing levels to control the listing in a way that allows the programmer to have local listing control without affecting the overall listing. When the listing level is negative, the assembler suppresses lines of code from the symbolic listing. The listing level is initially zero, and the `LIST` option increments the listing level while the `NOLIST` option decrements the listing level. Therefore, the user may bracket a program segment with a `LIST/NOLIST` pair with no net change to the current listing level for program statements. That is, `LIST` and `NOLIST` options at earlier points in the program remain in effect after the assembler has processed a subordinate `LIST/NOLIST` pair.

A similar option may be selected at run time, but the run-time option refers to the entire listing and cannot be alternately changed within the program the way the assembly-time options can (see Chapter refrunning-assembler-chapter for the corresponding run-time option discussion).

The default for the `LIST/NOLIST` option is `LIST`.

5.40.14 OPT LLRANGE

The `LLRANGE=n` option tells the assembler that n is the maximum number of lines that may separate a local label from any reference to the label. A local label can be used only within `n` where it is defined, or a local label range error (`??LR`) is generated. The default value of the local-label range parameter is 104 lines (or approximately two pages), but the user may change the unsigned value of this parameter as often as desired. Notice that the value zero effectively outlaws the use of local labels since a local label can not be accessed from the same line that defines it (see Chapter 2 for a discussion of local labels). The local-label range limit is a 16-bit unsigned value, so the assembler uses only the 16 least-significant bits of the specified value.

The distance between two lines is defined to be the absolute value of the difference between their line numbers. The distance from a line to itself is therefore zero.

The default local-label range limit is 104 lines.

5.40.15 OPT LONG/SHORT

The SHORT option tells the assembler to print only the first line of any statement (e.g., FCC, FCB, or FDB) that generates multiple output lines of object values, and the LONG option tells the assembler to generate all object lines of a multiple-line statement. This option allows the user to have only the significant line of a multiple-line generating instruction output to the symbolic listing (if one was selected), and shortens the listing by not listing lines that are not important. The SHORT/LONG options may be specified as many times as desired in the program.

The default for the SHORT/LONG option is LONG.

5.40.16 OPT LPAGE

The LPAGE=n option tells the assembler to produce n lines of output per logical page of the symbolic output listing. These n lines include the lines that the assembler automatically produces for the heading of each page. The assembler reports an invalid-option (IO) error unless LPMIN $\leq$ n $\leq$ PHYPG where LPMIN is somewhat arbitrarily defined to be 20 and PHYPG is the number of lines that exist from perforation to perforation on a physical page. The LPAGE parameter is initialized at the beginning of each pass to a value that provides top and bottom margins of one inch, but it may be changed as often as the user desires.

The default logical-page size is set according to the physical page size that the user specifies with the LPP run-time option. Since the default value for the LPP parameter is 66, the standard default value for the LPAGE parameter is $66 - 12 = 54$.

5.40.17 OPT MPERR/NOMPERR

By default, the assembler reports an error if the macro call does not specify enough parameter substitution strings. For a macro that uses positional parameters, an error is detected if the macro call contains fewer parameters than required. For a macro that uses formal parameters, an error is detected if any parameter that is not associated with a default string is not assigned in the macro call. The macro expansion processor substitutes a null string for any unspecified parameters.

Since the programmer may intend that the unspecified parameter(s) be replaced by the null string, the assembler allows the user to suppress reporting of the missing parameters error. The user should not routinely suppress the error message; it is best to suppress the error message to get a "clean" listing only after ensuring that nothing important is omitted.

The user can change between the report missing parameters mode and the suppress missing parameters error mode as often as desired by selecting the MPERR or NOMPERR option, respectively. There are also run-time MP and NMP options, but the OPT source-code options override the run-time options.

The default for the MPERR/NOMPERR option is MPERR.

5.40.18 OPT MPU

The assembler is now capable of assembling source lines as if they belonged to any of a number of M6800 closely related microprocessors. While the default `MPU` is the 6800, the user is able to specify that subsequent lines of source code are to be assembled for the following MPU's: 6801, 6802, 6803, 6808, 6809, 68HC11, 6301 and 65150. This option is also expressible at run time (see Chapter 6).

5.40.19 OPT MSHOW/NOMSHOW

The assembler shows macro expansion by default, but the user can change between the list mode and the no-list mode for macro expanded statements as often as desired by selecting the `MSHOW` or `NOMSHOW` option, respectively. When `MSHOW` is in effect, every line generated by the macro expansion processor (except for any `MEND`, `MEXIT`, or other instruction that is never listed) will be listed; when `MSHOW` is not in effect, only the macro call statement will be shown unless there are errors. Any macro expanded line that generates an error will be shown, regardless of the state of the `MSHOW` option. (Of course, the user can suppress all error messages with a run-time option.)

There is also a run-time `MS/NMS` option, but any `OPT MSHOW/NOMSHOW` source-code option overrides the `MS/NMS` run-time option. Refer to Chapter 6 for information on the run-time options.

The default for the `MSHOW/NOMSHOW` option is `MSHOW`.

5.40.20 OPT PAGENUM

The `PAGENUM=n` option tells the assembler to set the page number of the next page to `n`, and to restart the page numbering sequence from this point. This option statement is best used immediately before an `EJECT` statement. This option is also available as a run-time option (see Chapter 6) but the run-time option applies to the initial page number of the symbolic listing and it cannot modify page numbers in the listing after the first page. If both assembly and run-time options are selected, the page number will be reset at the occurrence of the `PAGENUM` option in the source stream as if the run-time option had not been selected.

The default value for the page number is 1.

5.40.21 OPT PARITY

The user may specify what parity an ASCII character in the program is to have with the `OPT PARITY=n` option. Currently only mark and zero parity are allowed. If `n` equals 1, we have mark parity, and if `n` equals 0, of course, we have zero parity. Any other value for `n` results in an invalid option (`??IO`) error. The option selected controls the values of the

characters in FCC strings as well as quoted character values in expressions. For example, the character "A" will have the value $C1 with mark parity, and an "A" will have the value $41 with zero parity. The user can switch between zero parity and mark parity values as often as desired.

Zero parity is assumed at the start of each assembly pass.

5.40.22 OPT RV8/NORV8

When NORV8 is in effect, the assembler issues a warning message for any 8-bit expression that could possibly exceed the 8-bit range at link time because of relocation and/or imports. When RV8 is in effect, the assembler doesn't issue such warnings, but the user will still get error messages (as appropriate) at link time.

NORV8 is the default.

5.40.23 OPT TABS

The "TABS=n1,n2,n3,...,nk" option tells the assembler to clear all previous tab stops and to set tabs in each specified column. Tab stops may be specified in any order, but an invalid-tab error (??IT) is reported unless $0 < nj < 81$ for each nj value specified. Note that all tabs can be cleared by setting a tab stop just in column 1 since a tab stop in column 1 is meaningless anyway. Unlike other OPT assembly options, no other options may follow a tabs option on the same line because the syntax assumes that all fields after a TABS option are expressions that define tab-stop columns.

Default tabs are set in columns 9, 17, 25, 33, 41, 49, 57, 65, and 73.

5.40.24 OPT TMERR/NOTMERR

By default, the assembler reports an error if the macro call contains too many parameter substitution strings. This error occurs when the macro call processor detects a comma following the last parameter substitution string of a positional-parameter macro call. The programmer may have tried to input more parameters than the macro supports.

Since the programmer may deliberately use a comma to delimit the final string as a shortcut to avoid using "<>" syntax to specify a null string, the assembler allows the user to suppress reporting of the too many parameters error.

The user can change between the report too many parameters mode and the suppress too many parameters mode as often as desired by selecting the TMERR or NOTMERR option, respectively. There are also run-time TM and NTM options, and the OPT source-code options override the run-time options.

The default for the TMERR/NOTMERR option is TMERR.

5.41 ORG

The `ORG` (Origin) statement sets the assembler's location counter equal to the value of the expression in the `ORG` statement's operand field. For example, `ORG $10` sets the location counter equal to `$10` (16 decimal), so the next word to be assembled will be assembled for location `$10`.

If a label is present in an `ORG` statement's label field, the label is equated to the value of the expression in the operand field. However, most good programmers don't include a label with an `ORG` statement because they can achieve more clarity by including the label with the statement that follows the `ORG` statement.

The assembler initializes the location counter to zero by default, so the location counter starts at zero if a user neglects to include an `ORG` statement at the beginning of a program. A good programmer, however, always includes an `ORG` statement at the beginning of a program for improved clarity and maintainability.

5.42 PAGE

The `PAGE` statement is equivalent to the `EJECT` statement, and the `PAGE` statement is included only for compatibility with Motorola's definition of the assembly language. Please refer to the description of the `EJECT` statement for more information.

5.43 PRINT

The `PRINT` statement is useful for displaying user-controlled messages in the listing file. In addition, these messages will appear at the user's terminal in the course of the assembly. The syntax of the `PRINT` statement is:

```
PRINT   expression      comment
```

When the assembler encounters this statement in the assembly, the assembler evaluates the expression and displays the expression value and the comment at the terminal during pass two. The assembler prints the value of the expression in both hexadecimal and decimal. If the assembler is generating a listing, the output from the `PRINT` statement will also appear in the listing. One possible use for this statement is in conjunction with conditional assembly. For example, if two mutually exclusive sections of code were both erroneously assembled in a large program, a thoughtful positioning of a `PRINT` statement could flag the user that an error in conditional-assembly logic had occurred. An error code of several values via the expression field could further pinpoint the exact nature of the error.

5.44 PSCT

The PSCT statement tells the assembler to start using the relocatable location counter that applies to the program section of memory, so the PSCT statement is allowed only in a relocatable assembly, not in an absolute assembly. Like the common section and the data section, the program section can be relocated to any memory address at link time, but the program section normally contains executable code and constants while the common section and the data section normally contain variables. For more information regarding the PSCT statement, please refer to Chapter 9.

5.45 RCB

The repeat constant byte, RCB, directive is a useful extension of the FCB statement. The count expression in the operand field defines the number of times that the assembler will perform the FCB function on the remaining expression values. The result is as if the corresponding FCB statement had appeared count times. The syntax of the statement is shown below, and further information on the acceptable values of the expressions is available under the FCB entry in this chapter.

```
label   RCB     count, value1, ...      comment field
```

5.46 RCBS

The RCBS directive is like RCB, but the RCBS statement specifies that the assembler should interpret and validate the expression value as a signed number.

5.47 RCBU

The RCBU directive is like RCB, but the RCBU statement specifies that the assembler should interpret and validate the expression value as an unsigned number.

5.48 RDB

The repeat double byte, RDB, directive is an extension of the FDB statement. The count expression in the operand field defines the number of times that the assembler will perform the FDB function on the remaining expression values. The result is much as if the corresponding FDB statement had appeared count times in the source file. The syntax of the statement is as shown below, and further information on the acceptable values of the expressions is available under the FDB entry in this chapter.

```
label   RDB     count, value1, ...      comment field
```

5.49 RMB

The `RMB` (Reserve Memory Byte) instruction is equivalent to the `BSS` instruction, and the `RMB` instruction is included only for compatibility with Motorola's definition of the assembly language. Please refer to the description of the `BSS` statement for more information.

5.50 SBTTL

The `SBTTL` assembler directive is very similar to the `SBTTLE` directive described in detail below, except that `SBTTL` does not solicit the `EJECT` function automatically. `SBTTL` spelled without the trailing "`E`" is mnemonic for no page "E"jection.

5.51 SBTTLE

The `SBTTLE` statement tells the assembler to use the string from the operand field as the current subtitle. The assembler always prints the current subtitle (if any) on the second line at the top of each page of the symbolic listing. The first line at the top of each page is used, of course, for the title line that is defined by the `TITLE` statement.

The user can naturally change subtitles as often as she or he wishes, so various sections of the same program can have different subtitles. If a `SBTTLE` statement is included as the first statement of a program (or at least before any statements that generate lines in the symbolic listing), the subtitle from that `SBTTLE` statement will be effective on the first page of the pass-2 symbolic listing. Otherwise, no subtitle line will appear on the first page of the pass-2 symbolic listing.

Since a new subtitle signifies the beginning of a new section of a program, the assembler automatically performs an `EJECT` function after it processes a `SBTTLE` statement. This feature eliminates the need for the user to include an `EJECT` statement immediately after every `SBTTLE` statement. Recall that an `EJECT` at the very beginning of a program has no effect since the listing is already at the top of a page then. Also recall that two or more consecutive `EJECT`s perform only one `EJECT` function, so the user who does include an `EJECT` immediately after a `SBTTLE` statement won't get a blank page. Similarly, a `TITLE` statement and a `SBTTLE` statement can be used together with no ill effects.

If the user specifies an empty subtitle string, the assembler eliminates the subtitle line entirely instead of printing a blank line in place of the subtitle line. Therefore, the listing maintains a uniform spacing style throughout an entire program regardless of the presence or absence of subtitles.

Most users employ the `SBTTLE` statement at the beginning of each new subroutine or at the beginning of each logical section of a program.

If the user has asked the assembler to generate a table of contents, the assembler uses all of the `SBTTLE` statements in the program to generate the table of contents. Please refer to the section that describes run-time options for more information.

If the first character of a title or subtitle string is a period, the assembler now throws that period away and does not print it as part of the title or subtitle. In the past you could not include any white space at the beginning of a title or subtitle string because the assembler assumed (as it still does) that the string started with the first nonblank character after the white space that terminated the opcode field. Now, however, you can effectively include leading white space in a title or subtitle string by using a period to start the string. Any white space that immediately follows the period will effectively become leading white space when the title or subtitle is printed.

You can use the new leading-period feature to generate a table of contents with indentation that graphically illustrates the logical structure of your program. The following example illustrates how you might want to use this feature to obtain indentation in your table of contents.

```
SBTTLE  .1.  Level 1
SBTTLE
SBTTLE  .    1.1.  Level 2
SBTTLE  .    1.2.  Level 2
SBTTLE
SBTTLE  .          1.2.1.  Level 3
SBTTLE  .          1.2.2.  Level 3
SBTTLE  .          1.2.3.  Level 3
SBTTLE
SBTTLE  .    1.3.  Level 2
SBTTLE  .    1.4.  Level 2
SBTTLE
SBTTLE  .2.  Level 1
SBTTLE
SBTTLE  .    2.1.  Level 2
SBTTLE  .    2.2.  Level 2
SBTTLE
SBTTLE  .3.  Level 1
```

The leading period in each subtitle string starts the string but is not printed in the listing, so the lines in the resulting table of contents are properly indented to show the logical organization of the program.

You can also use the leading-period feature to adjust the horizontal position of the title that is printed at the top of each page of the listing.

A circumflex (^) as the first character of a subtitle string tells the assembler that you want to start a new page in the table of contents with the current subtitle, so you can now control pagination in the table of contents. The leading circumflex character itself will not be printed as part of the subtitle string, but the remaining part of the subtitle string will

be at the top of a new page in the table of contents. If, for some reason, you want a literal circumflex as the first character of a subtitle string, you can simply start the subtitle string with a leading period (which will be ignored) and then type the circumflex after the period. In this case the circumflex will be a literal part of the subtitle string.

5.52 SET

The `SET` statement is similar to the `EQU` statement, but the `SET` statement defines a re-definable label. If you define a label with the `SET` statement, you can re-define that label as often as desired with other `SET` statements. An attempt to re-define a `SET`-defined label as a normal (i.e., not re-definable) label is not allowed. A `SET`-defined label is available for use from the time you define it until you re-define it to another value. A forward reference to a `SET`-defined label is therefore invalid. The assembler provides a separate concordance entry for each definition of a `SET`-defined label with multiple definitions, and the entries in each definition's reference list indicate references to that definition of the re-definable label. The assembler warns you if you try to export a re-definable label.

5.53 SETREG (6809)

The `SETREG` statement is analogous to the `SET` statement, but the `SETREG` statement takes a register list instead of an expression as its operand.

5.54 SK1

The `SK1` (SKip 1 byte) instruction tells the assembler to generate an instruction that will skip the next one-byte instruction during execution of the assembled (user's) program. By employing this instruction (as opposed to a branch instruction) the programmer will save one byte of memory. For example, the following instruction sequences execute the same way, but the code on the right uses one less byte of memory:

```
          .                        .
          .                        .
        BRA   1F                   SK1
SHIFT   LSRA             SHIFT     LSRA
1H      STAA  0, X                 STAA   0, X
          .                        .
          .                        .
```

The generated opcode value (`$85`) for a `SK1` instruction is the same value that is generated for the first byte of a two-byte `BITA` instruction with immediate addressing. When the `SK1` instruction is executed, the microprocessor takes the following byte as the second

byte of a BITA instruction with immediate addressing. The net effect is to skip one byte. As a side effect, the SK1 instruction may modify the N bit, the Z bit, and the V bit of the condition codes.

5.55 SK2

The SK2 (SKip 2 bytes) instruction tells the assembler to generate an instruction that causes the next two memory bytes to be skipped at execution time. The SK2 instruction is similar to the SK1 instruction, and it can be used to save the programmer one byte of memory. For example, the following instruction sequences execute the same, but the code on the right uses one less byte of memory:

```
          .                          .
          .                          .
         BRA   1F                   SK2
LOOP     ADDB  =4          LOOP     ADDB  =4
1H       STAB  2, X                 STAB  2, X
          .                          .
          .                          .
```

The opcode value ($8C) that is generated for a SK2 instruction is the same value that is generated for the first byte of a three byte CPX instruction with immediate addressing. When the SK2 instruction is executed, therefore, the microprocessor takes the following two bytes as the second and third bytes of a CPX instruction with immediate addressing. The net effect is to skip two bytes. The SK2 instruction may, therefore, be used to skip two single-byte instructions, or one double-byte instruction during program execution. As a side effect, the SK2 instruction may modify the N bit, the Z bit, and the V bit of the condition codes.

5.56 SKIP1

The SKIP1 assembler directive is just a more mnemonic name for the SK1 directive described above.

5.57 SKIP2

The SKIP2 assembler directive is just a more mnemonic name for the SK2 directive described above.

5.58 SPACE

The SPACE pseudo instruction is used to insert blank lines into a pass-2 symbolic output listing for improved readability. The SPACE instruction must not be labeled, and the value from its operand field tells the assembler how many lines to skip in the listing. For example, "SPACE 3" tells the assembler to leave three blank lines, and "SPACE N" tells the assembler to leave N blank lines, where N was previously equated to some number. Regardless of the ultimate operand value at least one blank line will be inserted into the listing.

The SPACE command is counted as a line, but it is not printed in the listing. Most programmers use the SPACE instruction to separate their assembly-language instructions into logical groups so their listing will be attractive and easy to read.

Users needn't worry about crossing a page boundary and generating any unnecessary blank lines at the top of the next page with the SPACE command because the assembler automatically stops skipping lines when it reaches the top of a page. A SPACE command that just happens to occur at the top of a page therefore does nothing. If a SPACE command occurs immediately after an EJECT command, however, the SPACE instruction generates the requested number of blank lines at the top of the page. This last feature allows a user to skip lines intentionally at the top of a page.

5.59 SPC

The SPC instruction is an abbreviated form of the SPACE pseudo instruction, which is described above.

5.60 TEMPLT

The TEMPLT directive allows the user to build a template for a data structure. The following example illustrates a simple use of the TEMPLT statement to build a template for a stack frame.

```
STKFRM  EQU     0                       The  TEMPLT statement's  argument
        TEMPLT  STKFRM                 specifies the initial value for the
SAVE_AR RMB     1                      location  counter  in  the template
SAVE_BR RMB     1                      block.  The initial value  for  the
RET_ADR RMB     2                      location counter is often zero.
```

A template defines the structure or organization of a data structure. A template does not reserve memory for the data structure that it defines. The assembler reports an error if you try to generate any object values in a template block. The next ORG or TEMPLT or SECT statement terminates a template block. Note that TEMPLT blocks are available in both absolute and relocatable assemblies.

A TEMPLT block is equivalent to the OFFSET block that is available with the 68000 family assembler. As a convenience for users with 68000 experience, the 6800 family assembler accepts OFFSET as a synonym for TEMPLT.

5.61 TITLE

The TITLE statement defines a title that is to be printed at the top of each page of any symbolic listing that is produced. An example of a TITLE statement is as follows:

```
TITLE Gas-Pump Controller Program (Version 3.7)
```

If this statement occurs in a program, the title "Gas-Pump Controller Program (Version 3.7)" will be printed at the top of every page of the symbolic listing until another TITLE statement is encountered.

The string that appears in a TITLE statement can contain as many characters as the size of a source line will allow. Furthermore, the user can change titles as often as desired in a program, so various sections of a program can have different titles.

The assembler automatically performs an EJECT function when it processes a TITLE statement, so the new section of the program automatically starts on a new page.

If a TITLE statement is included as the first statement of a program (or at least before any statements that generate lines in the listing), that title will be effective on the first page of the listing. Otherwise, the last title that was used during pass 1 will be effective at the beginning of pass 2.

The leading period feature as described in the detailed discussion of the SBTTLE directive above is also active in the TITLE statement.

5.62 VERIFY

The VERIFY statement provides a convenient method for a programmer to tell the assembler to verify some assumption that the programmer is making in the program. The VERIFY statement provides a convenient method for improving robustness and maintainability. The syntax for the VERIFY statement is as follows:

```
VERIFY <expression>
```

If the VERIFY statement's expression value is true (i.e., nonzero), the assembler simply lists the VERIFY statement in the program listing. If the VERIFY statement's expression value is false (i.e., zero), however, the assembler reports a VF error to alert the user to the fact that the stated assumption is false.

A user might employ the VERIFY statement as shown in the following examples:

```
        VERIFY  LIST_SIZE.LS.MAX_SIZE
        VERIFY  SPARES.GE.0
        VERIFY  TABLE_SIZE.GT.0
        VERIFY  TBL1SZ.EQ.TBL2SZ

        SKIP2
        VERIFY  2F-*.eq.2
1H      ADDA    =10
2H      INCB
```

5.63 XDEF

The `XDEF` statement is equivalent to the `EXPORT` statement, but the `EXPORT` statement is preferred.

5.64 XREF

The `XREF` statement is equivalent to the `IMPORT` statement, but the `IMPORT` statement is preferred.

Chapter 6

Running the Assembler

6.1 Getting Started

How many times has a new user of a software package been confronted with a thick user's guide and the dubiously helpful assurance from some more experienced user that "It's all in the manual ... somewhere"? Well, for just those people (including but not limited to harried students attempting to obtain a working program mere hours before it is due) the initial portion of this chapter on running the M6800 assembler is meant to provide the rudiments of assembly as directly and simply as possible.

Assuming that one is logged onto a VAX/VMS system and that a source file has been readied for assembly, then the command, `ASM`, executed at the VMS `$` prompt level will provoke the following response from the M6800 assembler:

```
FILES  (SRC,LST,OBJ):    here you type "sourcefilename" then <return>
OPTIONS:  here you type <return>
```

Now relax, but don't get too comfortable because the assembler is speedily processing the source file, cataloging and displaying at the terminal any errors (the message "`no errors in pass 2`" can be a welcome positive feedback) and generating the corresponding object and listing files. When the `$` prompt returns, the assembly is finished and the new files have been saved in the current directory as a hybrid between the "`sourcefilename`" and the extensions "`.obj`" and "`.lst`".

The listing file can now be printed with the VMS command `PRINT`, though it is a good idea to examine it with an editor first to verify its correctness (and thereby conserve paper resources). The object file is ready to be downloaded to a M6800 system, a possibly involved task which is best taught by example; though a subsequent section of this chapter attempts to describe in detail what is required.

For the tyro staring at the VMS $ prompt with despair, without even the source file assumed in the above discussion, consider trying the following incantation (as with all rituals, they must be reproduced exactly to obtain the desired results). The mysterious symbol <cr> stands for striking the carriage return key, and <cntrl-z> means the simultaneous depressing of both the <control> and <z> keys.

```
$ copy sys$input first.asm <cr>
        org     $200                            <cr>
alabel  ldaa    $100    ; a simple instruction  <cr>
        end                                     <cr>
<cntrl-z>
$
```

The file, first.asm, now exists in the current directory and it can be tried out in the role of a source file in the assembly process outlined above. As a harbinger to the novice, this example program includes an assembly error.

6.2 The Informed User

Since this user's guide describes a M6800 cross assembler written for the VAX/VMS operating system it seems apparent that any knowledge one has about the host environment would aid in potent use of this software. VMS documentation is notoriously complete and as a minimum it behooves the user to obtain at least a comfortable relationship with the following topics. If what is discussed below lacks clarity, the VMS on line help facility is suggested as a more complete explanation.

The concept of a file is of basic importance. In files one organizes information into predefined formats that pieces of software can manipulate. In particular, the file that contains the assembly statements is known as the source file. The M6800 assembler ingests a source file, which it never modifies, and as a function of its contents (as well as some run-time options) produces two files. The list or listing file contains a formatted representation of the source file more suitable for communicating the ideas of the source program. The list file also records any errors in the immediate proximity of the source line that caused them. The object file contains the ASCII hex packed machine code in S record format that corresponds to the semantics of the assembly source lines. The object file is suitable for downloading into a M6800 microcomputer for actual execution.

For software to access files, they must have unique names or identifiers. In VMS a complete specification of a file can consist of up to six parts. A full path name then is described by a node, a device, a directory (with possible sublevels), a filename (written without a space to signify a specific part of what in common parlance is a file name), a file extension, and a version number. Suffice it to say that the M6800 assembler supports

full file name specifications; however, most users will never need to be concerned with more than the filename and the file extension parts of an identifier.

For the file concept to be useful, the user must be able to create and rearrange them. Most files, like the assembly source files will be created with an editor like `EDT` on VMS. Though the initial contact with so many options and command keys may be intimidating, facile use of an editor will prove to be a quite valuable return on one's initial investment in learning to use it. Files, once created can be renamed, deleted, copied and printed. The pertinent VMS commands are `RENAME`, `DELETE`, `COPY` and `PRINT` respectively and are very easy to use (at times one may find the delete command disastrously easy). Again it should be stressed that `HELP` on any of these topics is just four characters away.

A word on the relationship between the version number part of the full file identifier and the `PURGE` command may save the user from generating arbitrary file names to maintain uniqueness for a sequence of versions in an evolving source program. A new file, when created, is given a version number of 1. If one edits this file and exits with changes using the same name, VMS automatically saves the current edition of the file with a version number incremented by one. Similarly, if the assembler is told to generate a listing or object file with a file name that already exists in the current directory, then the assembler creates these files with a version number one higher than the highest in the directory. Most VMS commands automatically use the highest version numbered file by default when the rest of the file name is common to more than one file, but the `PURGE` command deletes all but the highest version numbered file in such a set. The user should be aware, however, that most accounts are set up by default to automatically purge any files of a set in excess of the top three version numbers.

Files themselves are organized into structures known as directories (books organized on a shelf, or folders in a file cabinet drawer may be useful analogies). Most beginning users of VMS "wakeup" in their home directories when they login to the computer. If that is the directory from which the M6800 assembler is invoked, then any listing or object files will also appear in that directory; and, that is where the assembler searches for the file that was named as the input source. An educated user can easily override this default action by careful inclusion of the directory part of the complete file identifier.

A user can protect both files and directories with respect to such functions as reading, writing, and executing. The default protection that the system provides each user should be sufficient for most purposes. However, the user with sensitive information can more closely regulate his or her files using the `SET PROTECTION` command (as always, on line help is available for this command). Be advised that the assembler requires read permission on any source file that it is to process.

Generally, the M6800 assembler is invoked within an interactive session in VMS (other modes of operation are discussed in Section 6.6). In this situation, the user issues the `ASM` command as a response to the `$` prompt from the VMS interactive command line interpreter. (NOTE: the `ASM` command is specific to the VAX/VMS installation for the Digital Computer Laboratory at Arizona State University. More generally, the command `RUN M6800ASSEMBLER.EXE` must be issued to execute the assembler. `M6800ASSEMBLER.EXE`

is a generic name and it is the responsibility of the user at any given installation to discover the actual name in use there.)

Using the assembler as described in Section 6.1 often meets the needs of the user; however, the assembler is by no means limited to this rigid usage. Adopting the adage that one learns best by example, the following paragraphs show sample interactions with the assembler along with the resultant outcomes. To keep the examples simple it is assumed all files are in the home directory so that only file names and file extensions need to be considered. The examples are not exhaustive, and the user who still feels limited can refer to Section 6.6 in order to examine the parsing routine that processes the user's files and options specification input string. The symbol `<cr>` denotes striking the carriage return key of the keyboard.

```
FILES  (SRC,LST,OBJ):  LAB1   <cr>
```

This example repeats the context of the discussion in Section 6.1. The assembler attempts to access a source file named `LAB1.ASM` in the current directory. If this source file does not exist, or if the assembler is unable to read this file, a message alluding to a source file error is generated and the assembly terminates. When `LAB1.ASM` does exist, the file is assembled and the two companion files, `LAB1.LST` and `LAB1.OBJ` are created in the current directory. Generally, assembly errors found in the source are reported both to the terminal and within the listing file; though, these actions can be controlled with run-time options that are described later in this chapter.

By assuming the default file extension, `.ASM`, for source files, and by automatically appending `.LST` and `.OBJ` to the listing and object files, the assembler promotes the recommended organization files related to M6800 assembly on the VAX/VMS system. Not only will the user be able to save a few keystrokes in specifying files, but he or she will also accrue the advantage of simplifying communication with other users who adhere to the standard.

```
FILES  (SRC,LST,OBJ): sAM.SAM,Joe.SAM,JoE.JOE    <cr>
```

However, as this example shows, the user is not obligated to adhere to any logical scheme of naming the source, listing and object files. Although the assembler will not be confused, can the user say the same? Incidentally, VMS file identification is not case sensitive, which means that any combination of uppercase and lowercase alphabetic characters can be used in a file name. The assembler, of course, matches this attribute and will recognize the file `sAM.SAM` in the current directory even though VMS displays the name as `SAM.SAM` when a `DIR` command is issued.

```
FILES  (SRC,LST,OBJ):  LAB1,,NONE     <cr>
```

At times the user may only be interested in finding assembly errors in a source file currently being developed. In such a case, and for other situations as well, it may not be necessary to generate an object file. The assembler recognizes the special file name `NONE` (or for that matter, `noNE`, `none`, and all such permutations of case) to indicate that the corresponding file is not to be created. It is a good idea to specify `NONE` for a file when it will not be needed because it prevents wasting both computer time and disk allocation (especially for large files). The user can use this technique to suppress the listing file, the object file or both files for any given assembly.

This example also demonstrates that the three file names are fixed as positional parameters. The first position always belongs to the source file and must always be separated from the second position with a comma whenever action different from the default is specified. Similarly, the second position always corresponds to the listing file, and a comma must be used to separate it from the third parameter whenever other than the default action is specified.

After the assembly in this example (given that `LAB1.ASM` exists), the listing file, `LAB1.LST` will be created, but no object file will be generated. Because the action specified by `NONE` is not the default action for the object file, it was necessary to skip over to the third positional parameter by using the two comma separators.

```
FILES  (SRC,LST,OBJ):  LAB1,FINAL.LIS     <cr>
```

After the assembly given this specification, a listing file named `FINAL.LIS` and an object file named `LAB1.OBJ` will be created in the current directory. Note that it was not necessary to skip over to the third positional parameter by placing a comma after `FINAL.LIS` because the user specified the default action with respect to the object file.

```
FILES  (SRC,LST,OBJ):  ,LAB1.LST,LAB1.OBJ    <cr>
```

This specification results in the message, "`You must enter a source file name!`", and indeed that does seem to be a requirement for an assembly. The user will be prompted again for the file specification input string until a source file name is entered. The default action works only for the specification of listing or object files from a source file base, not in the opposite direction.

```
FILES  (SRC,LST,OBJ):  NONE,NONE,NONE    <cr>
```

This example, however, is not an error provided that the file, `NONE.ASM` exists in the current directory!

```
FILES  (SRC,LST,OBJ):  TT:,KB:,terminal    <cr>
```

One final file specification feature is demonstrated in this example. All three symbols (as well as their uppercase and lowercase permutations) are considered by the assembler to represent special nondisk "files". While the three symbols are truly synonyms (the user would probably only ever type the shortest), where they appear with respect to the positional parameters does make a semantic difference. For the rest of this discussion, any of these three symbols will be represented by `TT:`.

When `TT:` is specified for the source file, the assembler understands that the user is going to enter from the keyboard the entire source file one line at a time (with no editing once the carriage return at the end of each line is entered). In most situations, this would not be the preferred method of generating a source file since an editor is a far superior method for entering and modifying source line text. However, for the user who is both a careful, organized typist, and who is perhaps only testing one or two lines of questionable assembler syntax, the method will reduce the overhead involved with invoking an editor.

Because the assembler requires two passes at the source file to complete the assembly there is a side effect of this terminal-as-source mode of operation. Each line as it is entered is therefore copied to a source file named `TEMP.ASM` which resides on disk in the current directory. The assembler displays a message that it is creating this file before any lines are accepted to ensure that potential file name collisions are brought to the attention of the user. If for some reason the assembler is unable to create a file in this directory, the assembly is aborted and an error message is reported to the terminal.

The preferred method of ending this mode of source input is for the user to type a source line with the assembler accepted syntax of an `END` statement (a tab, the word "`END`" and a carriage return will work). Other methods of quitting this mode suffer from undesirable side effects. Stopping by entering a `<CONTROL-Z>` on a new line will always result in a missing end statement error; and, a `<CONTROL-C>` exit will also abort the entire assembly process (though the file `TEMP.ASM` will still reside on the disk). Users who terminate this mode with a `<CONTROL-C>` are employing the assembler as a very crude editor and should in preference consider the `COPY SYS$INPUT FILENAME` technique outlined for the inexperienced user in Section 6.1.

A different situation occurs when the special symbol `TT:` (or its equivalent) is specified in the input string as the listing or object files. In this case, the contents of the created file marked by the `TT:` symbol will be displayed on the user's terminal. While this may appear to be a useful technique for quickly examining the results of a short trial assembly, the user should be cautioned about several side effects of this mode of operation. First, if the file is displayed at the terminal, it will not be saved as a disk file in the current directory. Secondly, the lines of the displayed file will be interleaved with the messages that the assembler normally generates (e.g., errors in the source file). If both the listing and the object files are sent to the terminal the result will be a well shuffled deck of confusion.

Of course, the user is not required to replace all fields of the file specification with `TT:` or its equivalent, as in the above example. Any desired combination of the special symbols, `TT:`, `KB:`, `TERMINAL`, `NONE` and standard VMS file names may be employed to meet the needs of the user.

```
FILES  (SRC,LST,OBJ):  TT:,,NONE   <cr>
```

The user in this example desires to input the source file from the terminal one line at a time knowing that the source will be saved in a disk file, `TEMP.ASM` in the current directory. The user also desires a listing file which will be named `TEMP.LST`, and no object file will be generated.

6.3 Run-Time Options

After a user has successfully specified the files, the assembler outputs the prompting message "`OPTIONS:` " to solicit run-time options from the user. The user should respond by typing any desired option specifications followed by a carriage return. If the user wants the default settings for all run-time options, he or she can respond with just a carriage return. The default values of the options are the most-useful values, so users can often respond to the options prompt with just a carriage return. (A snapshot of the default state of the run-time option environment appears in the discussion of the run-time Help option subsequent to Table 6.1)

By specifying run-time options, a user can conveniently perform the precise assembly task that is desired for any given assembly, so the assembler doesn't waste any time or resources performing assembly options that aren't needed.

The option specifications are mostly one to three character mnemonics, and they may be entered in any mixture of uppercase and lowercase. The various option specifications may be typed in any order, but multiple option specifications must be separated from each other by commas. Spaces and tabs are permitted anywhere within the option request fields, but the option request line may not exceed 132 characters, and blanks and tabs are counted as input characters. The user is therefore warned against an extravagant use of blanks and tabs when numerous options are going to be selected.

The option mnemonics themselves have mandatory and optional entries. For example, to set the beginning line number of an assembly listing, the user may specify: `LIN = n`, `LINe = n`, `LINe n = n`, `LINe nu = n`, `LINe num = n`, `LINe numb = n`, `LINe numbe = n`, or `LINe number = n`. All of these entries are acceptable to the assembler for this option, as long as the mandatory characters "`LIN`" are included. The mandatory characters are shown in uppercase, and the optional characters are shown in lowercase.

Basically, two classes of run-time options exist. One class of options require a number to be input with them (e.g., line number, page number, lines per page, conditional-assembly

mask, etc.). For these options an equal sign (=) and, of course, a value form part of the required syntax.

The other class of run-time options is a group of Boolean flags. A user can select a Boolean option by simply typing the option name (or at least the capitalized significant portion of the option name). However, a user can "turn off" or deselect a Boolean option by prefixing the option name with the letter "N" or "n" to signify the negated action.

In any case, if an option is invalid or assigned an improper value, the assembler outputs an appropriate error message and then solicits new input with the "OPTIONS: " prompt again. All options that were correctly entered before the error was noted will remain in effect as specified, though the rest of the input line will have been ignored.

Effective use of the run-time options may be made more clear by the following suggestions. If a program is always run with the same options, and they are not the normal default options, it would be best if the user included these options in his or her source code. On the other hand, some options allow the user to format the output listing and to obtain different listings of the same source program by specifying the options prior to the program's assembly, and these are best specified at run time.

The available run-time options are tabled on the next few pages in alphabetical order for the user's convenience. Subsequent to the table appear more complete descriptions of each of these options. Those options that may also be selected in the source code via the OPT assembler directive are noted.

Table 6.1: Summary of Run-Time Options

Option	Remarks
ABSolute	Generate absolute object code. The assembler generates absolute object code by default, though relocatable assembly can be selected via RELocatable.
CAmask=hhhh	Set the conditional-assembly parameter to the 16-bit hexadecimal value, hhhh.
COMment, NCOMment	Include all comment lines in the listing output. This option does not apply to any comments that appear in the comment field of a source statement.
Cross, NCross	Generate a cross reference or concordance and include it in the listing output at the end of pass 2 of the assembly.
EOerr, NEOerr	Report expansion overflow errors that when a macro expansion exceeds the maximum line length in the model definition.
Error, NError	Report assembly errors to both the terminal and the listing file (but only once if they coincide).

Continued

Table 6.1 (Continued)

Option	Remarks
`FFeed, NFFeed`	Generate page breaks using the form feed character. This option should be used unless the output device does not support form feeds.
`Help, NHelp`	Inform the user of all the run-time options with a brief help message displayed at the terminal, but remain in the option gathering mode.
`IGNore, NIGNore`	Suppress source lines that start with a period in column one from the listing output.
`INdent=n`	Indent the left margin of the output listing by `n` character spaces. `INdent` and `Width` can be used to select the left and right margins of the listing.
`LCA, NLCA`	Lines associated with conditional assembly are to be included in the listing output.
`LINenumber=n`	Begin the assembly listing with the value `n` as the first line number.
`List, NList`	Include all source lines in the listing output. `NList` is used to suppress all lines except those that contain assembly errors from the listing output.
`LOC, NLOC`	Include the location counter field in the assembly listing output.
`LPP=n`	Set the number of lines in the physical page of the output device that receives the listing to `n`.
`MPerr, NMPerr`	Report an error if a macro call does not specify enough parameter substitution strings.
`MPU=xxxx`	Assemble the source files for the M6800 related microprocessor denoted by `xxxx`. For example, n=6800 represents the M6800 microprocessor (default value).
`MShow, NMShow`	Show the source lines generated by a macro expansion in the listing output.
`Page=n`	Number the first page of the assembly listing as `n`.
`RELocatable`	Generate relocatable object code.
`Report, NReport`	Report assembly errors to the listing output. This option does not affect errors sent to the terminal as selected by the `Error` run-time option.

Continued

Table 6.1 (Continued)

Option	Remarks
SEQuence, NSEQuence	Include the line number or sequence field in the listing output.
SLine, NSLine	Include the line number of definition for labels included in the symbol table.
SRC, NSRC	Include the source code field in the listing output.
Symbol, NSymbol	Generate a symbol table of all labels and their values, and include it in the listing output.
TMerr, NTMerr	Report an error if a macro call has too many parameter substitution strings.
Toc, NToc	Include a table of contents generated from SBTTL or SBTTLE statements in the listing output.
Value, NValue	Include the value or object code field in the listing output.
Width=n	Set the maximum number of columns in a line of the listing output to n.

6.3.1 ABSOLUTE

The assembler is capable of generating either absolute or relocatable object code provided that the programmer has taken care to observe the requirements of a particular mode with respect to label definition and usage. The absolute mode of assembly is selected with the ABSolute run-time option, and the companion mode is selected with the RELocatable option. More information on relocatable assembly can be found in Chapter 9 and in the discussions on the assembler directives, ASCT, BSCT, CSCT, DSCT, EXPORT, IMPORT and PSCT found in Chapter 5. The mode of assembly can also be controlled from within the source file using the REL and ABS switches with the OPT assembler directive.

The default assembly mode is absolute.

6.3.2 CAMASK

The conditional-assembly parameter may be set by the user to any 16-bit hexadecimal value with the CAmask=hhhh option. Only the low-order 16-bits of an input hexadecimal value are retained by the assembler, and overflow is not indicated to the user. The conditional assembly mask parameter is utilized in two ways. The primary use is through the special assembler token, .CAMASK. which is set to the 16-bit value of the run-time CAmask parameter. This token is then used explicitly in the expression field of the GOIF and GOUNLS conditional assembly statements. The conditional-assembly parameter is also used

implicitly in the less preferred mode of conditional assembly involving the `GOIFZ` and `GOULZ` assembler directives. Refer to Chapter 7 for more information on conditional assembly.

The default value for this option is zero.

6.3.3 COMMENT

The `COMment` option tells the assembler to include comment lines in the output listing. The `NCOMment` option suppresses comment lines from the assembly listing. This option is useful when comments are not needed or an abbreviated output listing is desired. In an academic or training environment, students may be given listings of code to figure out, and later given a commented version of the same listing. Note that this option does not affect comments that appear on an instruction source line.

The default is the `COMment` option.

6.3.4 CROSS

The `Cross` option tells the assembler to generate a cross reference listing and output it to the listing device/file at the end of pass 2 of the assembly. The `NCross` option suppresses the outputting of the cross reference listing, which also suppresses the undefined and unreferenced symbols listings. The cross-reference listing is a valuable tool for debugging a program, and it is almost mandatory for determining all the places in a program that refer to, use, or modify a memory location.

The default is the `Cross` option.

6.3.5 DEBUG

By using this run-time option, the user can include a symbol table in the generated object file. Each entry in the symbol table consists of an ASCII-string label and its corresponding numeric value. Thus, a debugging tool (e.g., MUDBUG) can help the programmer by providing labels in place of "magic" numbers. This debugging option is available in both absolute and relocatable modes.

6.3.6 EOERR

By default, the assembler reports an error if the expanded macro line overflows the source-line buffer (i.e., exceeds 126 characters). This error occurs when the length of the substitution string exceeds the length of the parameter name and the macro model line did not have enough unused spaces at the end to absorb the difference. For example, if `\1` is replaced by `ALPHA + BRAVO - CHARLIE/DELTA`, the resulting macro expanded line is going to contain more characters than the model line does.

Since the programmer may determine that nothing of value was lost in the expanded line (only the comment field was truncated), the assembler allows the user to suppress reporting of the expansion overflow error. The user should not routinely suppress the error; it is best to suppress the error to get a "clean" listing only after ensuring that nothing important is lost.

The `EOerr` option tells the assembler to report any expansion overflow errors, and the `NEOerr` option suppresses reporting expansion overflow. There are also assembler directive `OPT` switches, `EOERR` and `NOEOERR` which override the run-time options.

The default is `EOerr`.

6.3.7 ERROR

When the `Error` option is selected, error messages are sent to the terminal as well as to the listing. The `NError` option, on the other hand, suppresses errors from being sent to the terminal. If the terminal is the output listing device, this option is effectively ignored since errors are automatically reported to the assembly listing (see the `REPORT` run-time option for suppressing errors in the assembly listing).

`Error` is the default for this option.

6.3.8 FFEED

The `FFeed` option directs the assembler to issue form feeds rather than the slower carriage-return (`<CR>`) and line-feed (`<LF>`) combination. The `NFFeed` option directs the assembler to replace each form feed with an appropriate number of `<CR> <LF>` combinations. The selection of the `FFeed` option may speed up output processing considerably, especially for long listings. The `NFFeed` option is necessary when the listing file is to be listed at a device that does not support form feeds.

The default is the `FFeed` option.

6.3.9 HELP

The `Help` option can be selected by the user who, while in the context of the options processor, would like to be briefly refreshed on the default states and correct syntax of the run-time options. After the help message is delivered, the options prompt is again issued and another input string is processed. If an error was detected in the input string to the options processor, regardless of the state of the `Help` flag, this same action will occur. The default state of this option is to offer help only when requested or when a run-time option error has occurred. The help message, as it appears at the user's terminal follows.

```
These are the options available for the front-end interface.
```

```
With Parameters:  (Default values)

CAmask = (0)                    MPU = (6800)
INdent = (0)                    Page = (1)
LINenumber = (1)                Width = (132)
LPP = (66)

As Boolean State Flags:  (an N prefix toggles flag)

*ABSolute       *LCA            *SEQuence
*COMment        *List           *SLine
*Cross          *LOC            *SRC
*EOerr          *MPerr          *Symbol
*Error          *MShow          *TMerr
*FFeed           RELocatable    *Toc
*IGNore         *Report         *Value

Entries marked with "*" are active by default.
Help yields this message.
```

6.3.10 IGNORE

The `IGNore` run-time option controls the appearance of the source lines that begin with a period in column one in the listing output. Recall that these "dot commands" are often associated with text formatters that the user may employ to justify and align blocks of text in the listing. As the word "ignore" suggests, when this option is in effect, the dot lines do not appear in the listing. They may, of course, be included by specifying the negated variant, `NIGNore`, of this option. The analogous action within the assembler directive `OPT` regime is also possible with the `IGNORE` and `NOIGNO` switches.

The default action of `IGNore` is to suppress listing of period initiated lines.

6.3.11 INDENT

The `INdent` option allows the user to set the left print margin for his or her output listing. The listing may not be indented to such an extent that no meaningful output will result. Therefore, the maximum indentation allowed must provide for 29 print columns (i.e., $\texttt{INDENT} + 29 \leq \texttt{WIDTH}$). If the selected output width is 80 columns, for example, the maximum indentation is 51 since $80 - 29 = 51$. The `INdent` option, in conjunction with the `Width` option, allows the user to set the left and right output margins. It should be noted, however, that any line exceeding the right margin will be truncated. This feature is useful when the user wants assembled values and source code without any comments. For

example, a user might want to display the listing file at a CRT terminal with no comments for debugging purposes. To select the indentation the user specifies `INdent=n`, where `n` is an unsigned decimal number.

The default value for this option is zero (i.e., no indentation).

6.3.12 LCA

The `LCA` option tells the assembler to send any lines associated with conditional-assembly statements to the assembly listing. The `NLCA` option suppresses `GOIF`, `GOUNLS`, `GOTO`, `GOULZ` and `GOIFZ` statements, and statements with conditional-assembly labels, as well as conditionally skipped statements from the assembly listing. This option may also be specified in the source program by entering the `CALIST` or deselected by the `CANOLIST` pseudo-opcode instructions. The `LCA` or `NLCA` options are one-time selectable, but the `CALIST` and `CANOLIST` options may be employed in the source program to change between the list mode and the no-list mode for conditional-assembly statements as often as desired.

The default for the `LCA/NLCA` option is `LCA`.

6.3.13 LINE NUMBER

The `LINe number=n` option tells the assembler that the beginning line number of the assembly listing is `n`, where `n` is an unsigned decimal integer. Note that the largest line number that may be output is 65,535 after which the line number sequence will start over with zero. This option is convenient when writing modular program segments that will later be brought together with proper sequence numbering. Occasionally, blocks of line numbers are allocated before the start of a large programming project to ensure that confusion about the location of certain segments of code won't result during code merging. This option is also selectable in the source program through the use of the `LINENUM` option with the `OPT` statement, which will restart the line numbering sequence from the point of its occurrence in the program.

The default beginning line number is one.

6.3.14 LIST

The `List` option tells the assembler to send an assembly listing to the file or device (if any) that was specified for the listing output. The `NList` option, on the other hand, tells the assembler to suppress its output of the assembly listing. The `NList` option suppresses only the source lines that do not contain errors. All lines with assembly-language errors will still be sent to the listing file/device, if one was specified. The `NList` option is therefore very useful for obtaining a listing of all lines that contain errors, and the `NList` option is often specified when the terminal is selected as the output listing device.

The list and no list modes may also be selected in the source program by specifying the `LIST` or `NOLIST` options with the `OPT` statement. The `List` and `NList` run-time options are one-time selectable, but the `LIST` and `NOLIST` options with the `OPT` statement may be selected in the program to list or not list various segments of the program at the programmer's discretion (see Chapter 5 for a discussion of the `LIST/NOLIST` option).

The default for the `List/NList` option is `List`.

6.3.15 LOC

The `LOC` option tells the assembler to include the location-counter field in the assembler output listing. Conversely, the `NLOC` option instructs the assembler to suppress the location-counter field from the assembly listing. Suppression of this field has questionable merit, other than to narrow the listing width.

The default for the `LOC/NLOC` option is `LOC`.

6.3.16 LPP

The `LPP=n` option tells the assembler that the line printer (or terminal, as the case may be) has `n` lines per physical page, where `n` is a decimal integer greater than or equal to 32. The number of lines per physical page includes all lines from perforation to perforation. The top and bottom margins, therefore, are included in the physical page line count.

The assembler uses the physical page size to format the listing file correctly for its intended output device. If the listing file is printed on a device with a physical page size that is different from the physical page size that was specified in the `LPP` parameter, the listing pages will not match the physical pages and, if the `NFFeed` option is selected (see above) will eventually "walk" across the perforations. The user can avoid this undesirable situation by simply specifying the correct value for the `LPP` parameter.

The default value for the `LPP` parameter is 66, which corresponds to standard 11" paper with printing at 6 lines per inch. Since the assembler automatically provides top and bottom margins of six lines each, there are normally $66 - 12 = 54$ lines printed on each page. Two of these lines are always used for a title line and a blank line, and one more line is used for a subtitle if the user includes subtitles. Therefore, the assembler normally prints 51 or 52 lines of the user's program on each page.

6.3.17 MPerr

By default, the assembler reports an error if the macro call does not specify enough parameter substitution strings. For a macro that uses positional parameters, an error is detected if the macro call contains fewer parameters than required. For a macro that uses formal parameters, an error is detected if any parameter that is not associated with a default

string is not assigned in the macro call. The macro expansion processor substitutes a null string for any unspecified parameters.

Since the programmer may intend that the unspecified parameter(s) be replaced by the null string, the assembler allows the user to suppress reporting of the missing parameters error. The user should not routinely suppress the error message; it is best to suppress the error message to get a "clean" listing only after ensuring that nothing important is omitted.

The `MPerr` option tells the assembler to report any missing parameter errors, and the `NMPerr` option suppresses reporting missing parameters. There are also assembler directive `OPT` switches, `MPERR` and `NOMPERR`, which override the run-time option.

The default for the `MPerr/NMPerr` option is `MPerr`.

6.3.18 MPU

The assembler now has a run-time option for specifying the MPU for which you want to assemble the program. The format is `MPU=xxxx` where `xxxx` is 6800, 6801, 6802, 6803, 6808, 6809, 6301 or 65150. The default is 6800. The 6800, 6801, 6802, 6803, 6808 and 6809 are all Motorola products. The 6301 is produced by Hitachi, and it is an extension of Motorola's 6801 microprocessor. The option to assemble source lines for another but closely related microprocessor is also expressible within the source file using the `MPU` variant of the assembler directive, `OPT`.

The default microprocessor is the 6800.

6.3.19 MSHOW

The `MShow` option tells the assembler to show the program lines generated by macro expansion. The `NMShow` option suppresses listing of the expanded lines.

When `MShow` is in effect, every line generated by the macro expansion processor (except for any `MEND`, `MEXIT`, or other instruction that is never listed) will be listed; when `MShow` is not in effect, only the macro call statement will be shown unless there are errors. Any macro expanded line that generates an error will be shown, regardless of the state of the `MShow` option. (Of course, the user can suppress all error messages with the `NReport` and `NError` run-time options.)

This option may also be specified in the source program by entering the `MSHOW` option or deselected by the `NOMSHOW` option, and any such source-code options override any `MShow/NMShow` run-time options. Refer to Chapter 5 for information on the assembler directive, `OPT`.

The default for the `MShow/NMShow` option is `MShow`.

6.3.20 PAGE

The "`Page=n`" option specifies the initial page number for the symbolic output listing (if any). This option is convenient when a person is writing a program that is to be included as part of a report. The program listing can be generated to start at the correct page number of the overall report. For example, if the program listing is to begin on page 75 of a report, the user could type "`Page=75`" as a run-time option to tell the assembler to start the page numbering sequence with page 75.

For related information, please refer to the description of the "`OPT PAGENUM=n`" statement in Chapter 5.

The default value for the page number option is 1.

6.3.21 RELOCATABLE

The assembler is capable of generating either absolute or relocatable object code provided that the programmer has taken care to observe the requirements of a particular mode with respect to label definition and usage. The relocatable mode of assembly is selected with the `RELocatable` run-time option, and the companion mode is selected with the `ABSolute` option. More information on relocatable assembly can be found in Chapter 9 and in the discussions on the assembler directives, `ASCT`, `BSCT`, `CSCT`, `DSCT`, `EXPORT`, `IMPORT` and `PSCT` found in Chapter 5. The mode of assembly can also be controlled from within the source file using the `REL` and `ABS` switches with the `OPT` assembler directive.

The default assembly mode is absolute.

6.3.22 REPORT

The `Report` option tells the assembler to report errors normally in the assembly listing, and the `NReport` option tells the assembler to suppress the reporting of errors in the assembly listing. The `NReport` option can be used as a desperation measure to suppress diagnostic warnings for a final clean assembly listing, but it should be used with extreme caution since it can hide errors from the user. This option has no effect on the reporting of errors to the terminal as selected by the `Error/NError` option.

The default for the `Report/NReport` option is `Report`.

6.3.23 SEQUENCE

The `SEQuence` option tells the assembler to include sequence numbers (i.e., line numbers) in the assembly listing, and the `NSEQuence` option tells the assembler to omit sequence numbers from the listing. The selection of the `NSEQuence` option may be used to reduce the width of the listing, when necessary.

The default action is `SEQuence`.

6.3.24 SLINE

The `SLine` option tells the assembler to create and output a symbol table listing that includes an alphabetic listing of all nonlocal labels, each label's value, and its line number of definition. The `NSLine` option, on the other hand, suppresses outputting of the line numbers in the symbol table listing, if a symbol table output listing was requested (see the `Symbol` option). Hence to obtain a symbol table without the line numbers of definition, the user needs to select both the `Symbol` and the `NSLine` run-time options.

The default for the `SLine/NSLine` option is `SLine`.

6.3.25 SRC

The `SRC` option tells the assembler to output the user's source code to the assembly listing, and the `NSRC` option tells the assembler to suppress the user's source code from the listing. This option may be used to dramatically reduce the output listing width, and it is a valuable aid in determining how well someone understands the actual assembly code. This feature is definitely not recommended for any output listings you may wish to read again in the future.

The default for the `SRC/NSRC` option is `SRC`.

6.3.26 SYMBOL

The `Symbol` option tells the assembler to create and output a symbol-table listing, and the `NSymbol` option tells the assembler not to output a symbol-table listing. The symbol-table listing provides an alphabetic listing of all nonlocal labels used in the source program, and it also includes each label's value.

The default for the `Symbol/NSymbol` option is `NSymbol`.

6.3.27 TMERR

By default, the assembler reports an error if the macro call contains too many parameter substitution strings. This error occurs when the macro call processor detects a comma following the last parameter substitution string of a positional-parameter macro call. The programmer may have tried to input more parameters than the macro supports.

Since the programmer may deliberately use a comma to delimit the final string as a shortcut to avoid using "`<>`" syntax to specify a null string, the assembler allows the user to suppress reporting of the too-many-parameters error.

The `TMerr` option tells the assembler to report an error whenever too many parameters are detected, and the `NTMerr` option suppresses reporting these errors. There are also assembler directive `TMERR`, and `NOTMERR` switches in the `OPT` set, which override the run-time options.

The default for the `TMerr/NTMerr` option is `TMerr`.

6.3.28 TOC

The `TOC` option tells the assembler to output a table of contents for the program that is being assembled. The table of contents is derived from the `SBTTL(E)` statements that are included in the program, and the assembler prints the table of contents at the beginning of the listing. Each subtitle is listed in the table of contents along with the line number on which it occurs. The `TOC` option is especially valuable for large programs that contain many different parts. The `NTOC` option tells the assembler to suppress the table of contents. If no `SBTTL(E)` statements are included in the program, the `TOC` option is ineffective.

The default value for the `TOC/NTOC` option is `TOC`.

6.3.29 VALUE

The `Value` option tells the assembler to output the generated object code values to the assembly listing file or device, and `NValue` tells the assembler not to output the generated code values to the assembly listing. The `NValue` option may be selected to reduce the assembly listing width when the code values are not important.

The default for the `Value/NValue` option is `Value`.

6.3.30 WIDTH

The "`Width=n`" option tells the assembler that the maximum number of columns in an output line for the assembly listing is `n`, where `n` is a decimal integer in the range $\texttt{INdent} + 29 \leq \texttt{n} \leq 132$. Output lines are truncated when they exceed the specified number of columns. The `Width` specification applies only to the listing output and does not have any effect on the processing of the source line nor on the object module. The maximum number of columns allowed in any one input source line is fixed at 126 columns. Refer to the `INdent` option above for further information.

The default value for the `Width` option is 132.

6.4 After The Assembly

What happens to the source, listing and object files after the assembly is over is not really the province of a user's guide on the M6800 assembler; however, a few more paragraphs of suggestions may be appreciated by some users.

Generally, it is a good idea (to prevent possible confusion), once a project or source program has been finalized, to remove multiple copies and previous versions of the source file, and perhaps even to move the final version to another directory for safekeeping. Listing

files and object files, once they have served their immediate purposes, should no longer be kept in the directory since they take up valuable space and can easily be generated again from the source file if necessary.

As another possibly useful suggestion, the user is forewarned against the potentially vicious cycle of edit, assemble, debug ... during the development of a source program. Most users find that careful preparation and documentation, away from the machine, can greatly improve software productivity. The subconscious mind out on an evening walk can be an amazingly effective debugger.

While printing of listing files has already been covered, the discussion in Section 6.1 promised detailed instructions in downloading object files from VAX/VMS to a microcomputer and this information follows.

6.5 Downloading Object Files from VAX/VMS

The `TYPE` command in the VAX can be used to send an object file to the terminal for downloading into a microcomputer. To download an object file named `LAB5.OBJ` from the VAX into a microcomputer in the Digital Systems Lab at Arizona State University, proceed as follows:

1. Put the communications-control switch on the microcomputer into the `CPU` position, and sign onto the VAX. Use the same sign-on procedures that are used for any ordinary VAX run.

2. After the program has been assembled without any errors and the object file has been created, move the communications-control switch to the `MPU` position, and turn on the power to the microcomputer kit. Be sure the baud rate is set appropriately (300 baud if you're using a phone dial-up; 9600 baud for a direct port). Then push the reset button on the microcomputer to generate a reset (RSI) interrupt, and type "`LD.`" to initiate a MUDBUG load operation with a zero displacement.

3. Move the communications-control switch to the "load" position, and then type "`TYPE LAB5.OBJ`" at the terminal. After the VAX has received the carriage return at the end of the line, it will download the object program directly into the microcomputer. When the VAX has finished downloading the program, move the communications-control switch on the microcomputer back to the `CPU` position before proceeding with the VAX run.

6.6 Advanced Topics

Most users can ignore the material in this section, especially during a first reading. This section is included to provide the interested user an insight into the full capabilities of the M6800 assembler.

6.6.1 Parsing the Files and Options Specification Input String

Perhaps the singular case of the noun "string" in the title above will clue the reader that the Section 6.2 description is not complete with regard to the user's input specifications. In fact, a user is able to specify both the files and all the run-time options in a single input string in certain conditions that are outlined below.

Mere microseconds after the user enters the command, `ASM`, to execute the M6800 assembler, a call is made to a VMS system procedure, `LIB$GET_FOREIGN`, to recover any string that may appear on the same line as the `ASM` verb itself. This string, if it is not empty, is passed to the parser and is treated as if it were input in response to the files specification prompt. In order for this system call to function the `ASM` command must have previously been defined as a special string symbol (e.g., in the `LOGIN.COM` file at the user or group level) similar to that shown below, which is pertinent to the situation at the VAX/VMS installation at Arizona State University. The leading dollar sign is the critical feature of this string definition.

```
$ ASM :== $SYS$USERDISK:[UD.GROUP.OWNER]M6800ASM.EXE
```

In the discussion that follows white space is defined as any character that precedes "!" in the standard ASCII ordering (i.e., ASCII value `$21` is "!" definition automatically includes tabs (`$09`) and spaces (`$20`)). The parser recognizes either the comma (`,`) or the virgule (`/`) as a separator between tokens. Now, from the beginning.

If `LIB$GET_FOREIGN` returns with a null string the files specification prompt is issued and the user's input string is turned over to the parser. If the string is still empty or if it contains only white space before the first separator is found, no source file has been entered and the files prompt is continually re-issued (with an explanatory note) until a nonempty input is obtained.

The first token that appears in the input string is taken to be the source file. If no separator follows this token the default action, as described in Section 6.2, is taken with respect to the listing and object files, and then, the options prompt is issued and processed as usual.

If all that follows the first token is only a separator, then it is assumed that the default options are acceptable and the assembly begins immediately. Otherwise, the next action depends upon which separator is found and the next two paragraphs exhaustively trace the possible outcomes of each path of this bifurcation.

If the separator is a comma then the next token (with leading white space stripped) pertains to the listing file (though recall that as a positional parameter "comma white space comma" specifies default action for the listing file). If nothing remains in the input string the default action for the object file is taken and the options prompt is issued. If only a separator remains then the options prompt is foregone and assembly begins immediately.

If a token follows this separator it is associated with the object file. It is at this point where the level two description ends; however, if more characters are yet to be examined in the input string, they will be passed as tokens to the options processor.

If the separator is a virgule the remaining string is processed as option tokens and the default action is taken for both the listing and object files. Empty, white space or separator only strings that are passed to the options processor simple result in all of the default run-time options to remain in effect.

In all cases, once the options processor is invoked as a result of the first input string, the options prompt will only be issued again due to an error or at the user request for help. The next topic on alternative modes of assembler execution may help explain the reasoning behind the parser as it has been implemented in the M6800 assembler.

6.6.2 Alternative Modes of Assembler Execution

The Section 6.2 description of the assembler execution focused upon the interactive mode. In this section, two other modes of assembler operation, within a command file, and as a submitted job to the batch queue, are discussed by example.

A command file is a sequence of commands that normally would be entered at the $ prompt in a VMS interactive session. The advantage of command files is that often repeated multi-stage tasks can be executed by simply typing one command file name. In some sense, a command file can be viewed as a macro to the command line interpreter of the VMS interactive shell. Command files are also used to tailor the environment in preparation for the execution of a particular program. Consider these two examples.

```
$ ! This command file is named DEBUG.COM
$ !
$ ! Exclamation points are comment lines.  Command files are
$ ! executed by prefixing the file name with an @ sign.
$ !
$ ON ERROR THEN EXIT
$ !
$ ! If something goes wrong terminate this command file, a useful
$ ! command for new users of command files.  Also, the assembler
$ ! returns an exit code via a system exit call so that the command
$ ! file is able to detect errors in assembly.
$ !
$ ASM 'P1',TT:,none
Nlist
$ !
$ ! The P1 in quotes represents the first parameter passed to the
$ ! command file as:       @DEBUG.COM  FILENAME
$ ! Note that the options are taken from the input stream of
```

```
$ ! this command file.  Data records have no leading $ sign.
$ ! This command file is set up to run the assembler for a file in
$ ! the debugging stages when only errors are to be reported to the
$ ! terminal.
$ !
$ EXIT
```

This next example is a command file named `FINAL.COM` and is used to tailor the assembler for a source file once it has been debugged and finalized. Notice that all of the arguments are listed on the `ASM` command line which `LIB$GET_FOREIGN` will pass to the assembler.

```
$ !  This is FINAL.COM and is executed as @FINAL filename
$ ASM  'P1' / NEO,NMP,NTM
$ EXIT
```

Of course, these two samples just hint at the possibilities of command file operation used in conjunction with the M6800 assembler. Furthermore, once command files are understood, they can be used with the VMS submit command as jobs in the batch queue. For example, if an assembly job is quite involved and expensive the user can usually reduce costs by submitting the job to run during off peak hours. A typical submit command for `SAVINGS.COM` would look as follows:

```
$ SUBMIT/NOPRINT/AFTER=13-JAN-1986:02:35 SAVINGS.COM
```

The `NOPRINT` option keeps the log file, which captures all output from batch job, from being printed and deleted. The `AFTER=date:time` parameter specifies when the job is to be executed. Parameters can also be passed to batch files and the interested user is referred to the extensive literature.

The wary reader may wonder what would happen if the special assembler file name, `TT:` (or its equivalent) is specified as the source file within a command file or a batch job submission. VMS clearly answers this question by defining two logical labels `SYS$INPUT` and `SYS$OUTPUT`. The assembler always gets information from `SYS$INPUT` and this includes the source lines when `TT:` is specified as the source file. The assembler always outputs information to `SYS$OUTPUT`. In interactive mode, both of these system symbols refer to the user's terminal as was discussed in Section 6.2. For a command file, `SYS$INPUT` is the command file itself (see `DEBUG.COM`) but `SYS$OUTPUT` remains the terminal. In a batch job, `SYS$INPUT` is again the submitted command file, but `SYS$OUTPUT` becomes the log file that is associated with each batch process. Hence, one can only advise the user against using `TT:` as the source in a command file or batch job.

6.7 Final Notes

The M6800 assembler should not be viewed as a fixed and finished software product. This assembler has history, and it has a future; it is an ongoing process. While the claim is made that the assembler is fast and robust, it is the intent of the author that future growth is in the direction of better meeting the needs of the user. For this reason it is hoped that the user of this assembler will take the time to explore its features, and what is more important, will offer suggestions for its improvement.

Of particular interest are documented examples of assembler behavior that seem to operate contrary to that outlined in this user's guide. Any report of a combination of files and options that cause the assembler to exit in any but a controlled fashion, will be promptly acted upon and any changes that are necessary will be conscientiously completed.

The Assembler is Robust...

As an exercise, the M6800 assembler itself, which is written in Macro-32 assembly language for the VAX, was run through the assembler as a source file. In the course of ninety seconds of execution, the assembler found 21,748 errors with itself! For that matter, the binary executable image of the M6800 assembler was also assembled and 53 errors were reported after five seconds of execution.

The Assembler is Fast...

When it wants to be! (Felix qui potuit rerum cognescere causas)

The Assembler is Growing...

The set of run-time options is being extended to more completely compliment those selectable via the assembly `OPT` statement. At this time a more comprehensive interactive help facility is also being considered.

Chapter 7

Conditional Assembly

The M6800 assembler provides the user with conditional assembly capabilities so that a single source file can produce multiple versions of a program. Conditional assembly allows the user to include or omit blocks of source lines automatically and dynamically, based on the logical verity of an arbitrary expression that can reference a 16-bit conditional-assembly parameter. The assembler supports two separate protocols for conditional assembly. Though the most recent version is preferred, this chapter documents both.

Conditional assembly greatly eases program maintenance because it allows people to maintain only one source file of a program, even if the program exists in several different object versions. Programmers who must keep several different versions of a program up to date frequently forget to edit the latest changes into every source version of the program, but conditional assembly eliminates this possibility of errors and frustration by making it unnecessary to keep more than one source element for any given program.

Beginning programmers seldom use conditional assembly, except as an exercise, but systems programmers use it extensively. Software houses commonly employ conditional assembly to produce different object versions of their software to match the various hardware configurations that exist in the field. They code one master source program with blocks of conditionally assembled code in it; and when they assemble the program for a specific computer configuration, they simply specify a conditional-assembly parameter in accordance with the configuration output desired. The assembler automatically assembles the proper blocks of code, and the resulting program runs on the desired hardware configuration.

As a specific instance of this situation consider the M6800 assembler itself written for the VAX/VMS operating system using the MACRO-32 assembler. Because the 6800 and 6809 microprocessors are very similar, a M6809 assembler was essentially contained within the existing M6800 assembler. Any changes with respect to the slightly different opcodes and addressing modes between the two microprocessors are compensated for with judicious use of conditional assembly. This selective inclusion of opcode tables, legitimate expression tokens, and program segments relevant to specific addressing modes can easily be made by toggling the value of two constants that appear at the very beginning of the assembler source

code. (In this respect, the MACRO-32 assembler is not as flexible as the M6800 assembler since no dynamic parameter can be specified. The fact that the value of the conditional-assembly parameter is set dynamically by the user is important because it permits the user to assemble multiple versions of the same program from the same source file without editing the file.)

For the M6800 assembler, the conditional-assembly parameter is set via the `CAmask=n` run-time option (confer Chapter 6). If no value is specified by the user for the conditional-assembly parameter, the assembler uses a default value of zero. Depending upon the mode of conditional assembly, this parameter can interact with the expression in a conditional-assembly statement in two ways.

The first is directly through the special assembler-recognized token, `.CAMASK.`, which always takes on the value of the run-time parameter, `CAmask`. The `.CAMASK.` token may appear anywhere in an arbitrary expression on the conditional-assembly statement source line. This method is the newer and clearer (and hence to be preferred) mode of conditional assembly supported by the M6800 assembler. The pseudo opcodes `GOIF` (go if true) and `GOUNLS` (go unless true) are used to skip over or to include conditionally assembled blocks of code.

The second mode uses the conditional-assembly parameter in a more rigid manner by always evaluating the logical AND of the `CAmask` run-time value and the 16-bit value of the expression from the conditional assembly statement in the source line. This mode of conditional assembly, which will eventually be phased out of operation (though is still being supported for compatibility reasons) uses the pseudo opcodes `GOIFZ` (go if zero), and `GOULZ` (go unless zero) to skip over or to include conditionally assembled blocks of code.

Regardless of the mode of conditional assembly, the `GOTO` pseudo instruction is employed to skip from one block of conditional-assembly code to another. While the `GOIF` and `GOUNLS` statements depend on the value of an expression and the `GOIFZ` and `GOULZ` statements depend on the value of an expression ANDed with the conditional-assembly parameter, the `GOTO` statement is evaluated unconditionally.

Conditional assembly also uses system labels. A system label is not entered into the symbol table, and it is uniquely identified with a dollar sign ($) followed by a hexadecimal value. Since system labels are not put into the symbol table, the same system label can appear several times in one program. System labels are used as targets for the `GOIF`, `GOUNLS`, (or `GOIFZ` and `GOULZ`) and `GOTO` statements. Conditional-assembly statements are normally listed in the pass-2 symbolic listing, but they may be suppressed from the listing either with a run-time option (confer Chapter 6) or with an `OPT` pseudo instruction as discussed in Chapter 5 of this user's guide.

Perhaps the functions of conditional assembly can best be understood through the use of examples. "`GOIF XX, $10`" means: If the expression `XX` is true (i.e., evaluates to a nonzero value), then skip over the intervening source lines until reaching the target label, `$10`. If `XX` is false (i.e., zero), then the assembler includes the intervening source lines in the assembly. The statement "`GOUNLS YY, $200`" means: If the expression `YY` is false (i.e., zero), then

skip over the intervening source lines until reaching the target label, $200. If YY is true (i.e., nonzero), then the assembler includes the intervening source lines in the current assembly. Notice that the assembler does not require XX and YY to use (either explicitly or implicitly) the special run-time conditional-assembly parameter token, .CAMASK.; the fact that XX and YY can use .CAMASK., however, makes conditional assembly a dynamic feature that the user can control at assembly time.

The GOTO statement is an unconditional skip statement, and the assembler's action does not depend upon the evaluation of any expression. For example, "GOTO $50" will always result in skipping source lines until a statement with the system label $50 occurs.

Similarly, we can explain GOIFZ and GOULZ with some examples. "GOIFZ XX, $10" means: If the expression XX ANDed with the value of the conditional-assembly parameter yields zero, go to the statement labeled $10 (i.e., skip all source lines until finding the system label $10). If the expression XX ANDed with the conditional-assembly parameter does not yield zero, the GOIFZ statement is effectively ignored. Similarly, the statement "GOULZ YY, $200" means: If the expression YY ANDed with the conditional-assembly parameter is not zero, skip source lines until a source statement that has the system label $200 is found. If the result of the YY expression value ANDed with the conditional-assembly parameter was zero, the GOULZ statement would have been ignored.

As a general rule, a target for a conditional-assembly statement cannot appear before the conditional-assembly statement in the source program because the assembler skips only in a forward direction. Furthermore, since the assembler must know in pass 1 which statements it is to assemble, the expression values in the GOIFZ and GOULZ statements must be defined in pass 1. Therefore, the expressions in the operand fields of these instructions may not contain any forward references.

The following example illustrates the use of conditional assembly with the preferred mode, in a program segment. Of course, conditional assembly could also be used to selectively define tables of constants in a data segment corresponding to the specific device driver code being assembled in the example.

```
TAPE    EQU     1.ROL.3          Bit 3 of the conditional-assembly
                                 parameter corresponds to the symbol TAPE.

DISK    EQU     1.ROL.4          Bit 4 of the conditional-assembly
                                 parameter corresponds to the symbol DISK.

        .                        This block of code is common for the
        .                        three versions of the program that may be
        .                        produced in this program segment.

        GOIF    DISK.AND.-       If bit 4 of the conditional-assembly
                .CAMASK..EQ.0,-  parameter is zero, skip statements until
                $770             finding the system label $770.
```

```
        .                       This block of code is assembled only if
        .                     bit 4 of the conditional-assembly parameter
        .                     is set.  Otherwise, it is skipped.  This
        .                     code therefore corresponds to the DISK
        .                     version of the program.

        GOTO    $1000           Skip the other versions.

$770
        GOIF    TAPE.AND.-      If bit 3 of the conditional-assembly
                .CAMASK.,-    parameter is set, skip all statements until
                $900          finding the system label $900.

        .                       This  block  of code is assembled only if
        .                     bits 3 and 4 of the conditional-assembly
        .                     parameter are zero.  This code therefore
        .                     corresponds to the nonDISK and nonTAPE
        .                     version of the program.

        GOTO    $1000           Skip the TAPE portion of the program.

$900                            This block of code is assembled if bit 4
        .                     of the conditional-assembly parameter is
        .                     not set, but bit 3 is set.  This portion
        .                     therefore corresponds to the nonDISK, TAPE
        .                     portion of the program.

$1000                           This is a dummy target line for all three
        .                     routines to enter, and the rest of the
        .                     program consists of more common code.

        END
```

Since the conditional-assembly parameter contains 16 bits and the `GOUNLS` and `GOIF` pseudo ops involve arbitrary operators (not necessarily `.AND.` as in the example) in expressions, several different conditional variables can appear in the same assembly; and conditional assembly can become quite a bit more complex than this simple example indicates.

For the sake of completeness, the critical lines of the example are repeated using the older and less preferred mode of conditional assembly. In this mode the interaction of the conditional-assembly parameter with the expression in the source line statement is implicit and hence more difficult to document and maintain.

```
        GOIFZ   DISK, $770      If bit 4 of the conditional-assembly
                                parameter is zero, skip statements until
                                the system label $770 is found.

$770
        GOULZ   TAPE, $900      If bit 3 of the conditional-assembly
                                parameter is set, skip all statements until
                                the system label $900 is found.
```

Chapter 8

Macro Assembly

The M6800 assembler provides the user with macro assembly capabilities so that a single source-code line can generate multiple program lines. Macro assembly allows the user to define a commonly used sequence of instructions only once, then insert that sequence of instructions by naming the macro. The macro processor is basically a text-manipulating tool that works in conjunction with the rest of the assembler.

A macro definition, which is introduced by the `MACRO` pseudo-op, consists of a model call statement and a series of model statements followed by the `MEND` pseudo-op. The macro definition can optionally contain parameters. When the macro is later called, the parameters in the macro definition will be replaced by the actual strings specified by the user in the macro call.

A macro is called by simply naming the macro in the opcode field. The macro processor copies the lines of the macro model, one at a time, as though they were part of the source file. The assembler replaces any occurrence of the parameters with the actual substitution strings specified in the macro call. The process of copying the macro model lines and replacing the parameters is called "macro expansion", and the lines produced by macro expansion are called macro-expanded lines.

The macro processor provides the following features:

- Positional or formal parameters
- Default strings for formal parameters
- Nested macro calls
- String concatenation
- Automatic label field generation
- Delimited or undelimited strings

- Continuation of undelimited strings
- User-controlled listing or suppression of expanded lines
- Syntax error reporting (some of which can be suppressed)

The following shows the structure of a macro definition.

```
MACRO
<macro name>    [<formal parameter list>]
<model
statements>
MEND
```

8.1 Macro Names

A macro name consists of an alphanumeric string that must begin with an alphabetic character. Unlike normal symbol names, only the first six characters of a macro name are significant. This restriction is consistent with the fact that macros are used like user-defined opcode mnemonics, which also contain only six significant characters. Also unlike normal symbol names, macro names can be redefined as often as the user desires, and a macro call will result in expansion of the most recent definition. If a macro name is the same as one of the native opcode mnemonics, the macro has precedence over the opcode. Note that macro names are stored in a different list from normal symbols, so the program can define and use a macro of the same name as a label without any conflict or ambiguity.

The macro name belongs in the opcode mnemonic field. In the macro definition, the macro name should be in the opcode field of the first line following the `MACRO` pseudo-op, which is described in Chapter5. The line with the macro name must not be labeled. Besides the fact that a label would be meaningless, the assembler would interpret the label field as the macro name, and the macro name as a formal parameter name.

8.2 Macro Parameters

The macro assembler recognizes two types of parameters. Formal parameters are named, and positional parameters are recognized only by their position in a macro call. A macro can have no parameters, positional parameters, or formal parameters, but formal and positional parameters are not recognized for the same macro.

Regardless of the type of parameter used, the assembler will replace each occurrence of a parameter name in the macro model with the substitution string specified in the macro call.

If the macro call model statement in the definition contains a parameter name within `CMTCOL` columns after the macro name, then the macro uses formal parameters. (See Chapter 5 for an explanation of the `CMTCOL` option.) If no alphabetic character appears within `CMTCOL` columns following the macro name, then the macro uses positional parameters.

8.2.1 Positional Parameters

Positional parameters do not have formal names, but are identified by an index of the form `\n`, where `n` denotes the position of the parameter in the macro call. Parameter index values must be in the range 1 through 255. `\1` denotes the parameter that is associated with the first parameter, `\2` denotes the second parameter, and so forth. The user simply scripts a `\n` index wherever a parameter string is to be substituted.

When the assembler stores the macro definition, it determines how many parameters the macro uses, i.e., the largest index value. Then, when the assembler encounters a macro call, the macro processor expects to find that number of strings. The assembler reports an error if there are too few or too many parameters. The missing parameter and too many parameters messages may be suppressed using the source-line `OPT` switches, `NOMPERR` and `NOTMERR` or the run-time options `NMP` and `NTM`. Refer to Chapters 5 and 6 for information on the use of options and refer to Chapter 10 for information on error messages.

The macro call model line in the definition of a positional-parameter macro must not contain anything within `CMTCOL` columns following the macro name.

The following simple example illustrates the use of positional parameters.

```
MACRO                    Example usage of positional parameters.
TEST
ADDA    \1               Add \1 to the A register and store the
STAA    \2             result in \2.
MEND
```

When the assembler reads the definition above, it stores the information needed to handle macro calls for the macro named `TEST`. The assembler saves the model lines and the number of parameters (2).

The following illustrates a macro call and the resulting expansion. String syntax is explained later in this chapter.

```
TEST    COUNT, <0, X>
ADDA    COUNT            Add COUNT to the A register and store the
STAA    0, X           result in 0, X.
```

When the assembler encounters the macro name (`TEST`) in the opcode field, it gathers the parameter substitution strings from the call and substitutes those strings into each line of the model as the assembler expands each line. Notice that wherever the assembler found `\1` in the model, it substituted the first string (`COUNT`), and `\2` was replaced by the second string (`0, X`). It does not matter what field the positional parameter index appears in. In this example, parameter indices appear in the operand field and in the comment field.

8.2.2 Formal Parameters

The use of formal parameter names can make the macro definition easier to read and provides the capability of specifying default parameter substitution strings. A parameter name is an alphanumeric string that must begin with an alphabetic character. All characters of the name are significant, regardless of the setting of the `LBLLIM` option. Since the parameter names are associated with a particular macro definition, the same parameter name can be used in more than one macro definition with no conflict or ambiguity. The user can also use the same name for a parameter and a normal symbol, but that may cause an unexpected result. The macro processor will substitute for any parameter it finds in the macro model. For example, if the macro definition defines `X` as a parameter name, and a model line references the variable `X`, the assembler will replace the variable `X` with the specified string.

As the following example illustrates, formal parameters are used in much the same way as positional parameters except that they are explicitly named. Note that the first parameter name must begin within `CMTCOL` columns after the macro name. The example shows the use of a generated label field, which is explained later in this chapter.

```
        MACRO                     Clear block; shows use of formal
        CLRBLK  FROM, TO        parameters.

        STX     SAVE_X            Save the current value of the XR and
        LDX     =FROM           load the address of the first location in
                                the block.

LOOP_|# CLR     0, X              Clear each memory location and advance
        INX                     the index register to address the next
        CPX     =TO+1           location.  Loop until the last location
        BNE     LOOP_|#         has been cleared.

        LDX     SAVE_X            Restore the XR.
        MEND
```

The following illustrates a macro call and the resulting expansion.

```
        CLRBLK  TAB1, -           Clear table 1.
                TAB1+TABSIZ-1

        STX     SAVE_X            Save the current value of the XR and
        LDX     =TAB1           load the address of the first location in
                                the block.

LOOP_A  CLR     0, X              Clear each memory location and advance
```

```
INX                     the index register TAB1+TABSIZ-1 address th
CPX     =TAB1+TABSIZ-1+1                location.  Loop until the l
BNE     LOOP_A

LDX     SAVE_X            Restore the XR.
```

Notice that wherever `FROM` appears in the model, the assembler substituted `TAB1`; and wherever `TO` appears in the model, the assembler substituted `TAB1 + TABSIZ − 1`. The assembler even used the substitution string where the word "to" appears in the comment field. The user should choose parameter names carefully to avoid such unintentional substitutions.

Expansion overflow occurred on two lines because the parameter string was much longer than the parameter name. In both cases, only the comments were truncated, but the user should check expansion overflow errors to ensure nothing of value has been lost. Refer to the discussion in Chapter 10 regarding the `EO` error message and in Chapters 5 and 6 regarding suppression of the expression overflow error message.

There is nothing magical about the use of underscore (_) in the names `SAVE_X` and `LOOP_|#` in the example, but it does illustrate a technique the user may want to consider. Underscore is one of the special characters that the assembler treats as alphabetic. The use of some special character on labels associated with macros may help avoid conflicts with other symbols. For example, successive expansions of macro `CLRBLK` will define labels `LOOP_A`, `LOOP_B`, etc. The use of the special character, along with some other naming conventions, can prevent conflicts with `LOOPA`, `LOOPB`, or other labels the user has defined directly. Information on automatic label field generation is presented later in this chapter.

One advantage of formal parameter names is that they provide the ability to specify default substitution strings. For example, if the earlier definition were modified as shown below, the user could call the macro without having to specify the parameters. Assume that the most common use for macro `CLRBLK` is to clear an 80-byte buffer named `BUFF`. We might change the first part of the definition to specify:

```
MACRO
CLRBLK  FROM = BUFF, -
        TO = BUFF+79
```

Now any macro call that does not expressly specify substitution strings for `FROM` and `TO` will be equivalent to listing the default strings in the call. Default values can be quite convenient when a macro habitually performs the same specific task.

8.2.3 Parameter Substitution Strings

A string must satisfy one of the following definitions:

- Any group of characters not containing any blanks, tabs, carriage returns, or commas, and not beginning with <, ^, or a hyphen followed by blanks or tabs. This last situation will be interpreted as line continuation, so the hyphen and any characters that follow the hyphen will be ignored, and the first nonblank character on the next line will begin the string.

```
Examples:       X

                ALPHA/BRAVO+CHARLIE.MOD.DELTA

                ALPHA-
                +BETA

                1$
```

- Any group of characters (or a null string) preceded by < and and followed by > on the same line.

```
Examples:       <This is a string.  >

                <>
```

- Any group of characters (or a null string) delimited by a nonblank character of the user's choice. The opening delimiter must be preceded by an up-arrow or caret (^), and a matching delimiter must close the string on the same line.

```
Examples:       ^@So is this.  @

                ^^Even this is O.K.  ^

                ^//

                ^#!"$%&'(){}|\ ?><][:;*__#
```

Notice that strings fall into two general categories: delimited and nondelimited. Nondelimited strings may be continued over more than one line, and the assembler will copy the string onto successive lines. For example, we could have used continuation in the last example to minimize expansion overflow. If the call had been

```
CLRBLK  FROM = TAB1, -
        TO = TAB1-1-
        +TABSIZ
```

the 8th line of the macro model would have been expanded as

```
CPX     =TAB1-1-
        +TABSIZ+1
```

The assembler supports continuing an expression (string) over any number of lines.

8.3 Macro Model Statements

Each line of the macro model can contain characters that are to be copied verbatim, parameter names (or parameter indices for positional parameters), concatenation operators (`|`), and automatic label fields (`|#`).

8.4 String Concatenation

It is sometimes desirable to concatenate substitution strings. The concatenation operator (`|`) is an "invisible" delimiter. A single vertical bar in a macro model line is never copied to the expanded line. Instead, the character is simply skipped. For example, consider the following simple macro.

```
MACRO
CONCAT  LEFT, RIGHT
ASCII   "LEFT|RIGHT"
MEND
```

The following call and expansion illustrate the effect of the concatenation operator.

```
CONCAT  LEFT = DOWN, -
        RIGHT = TOWN
ASCII   "DOWNTOWN"
```

The concatenation operator delimits the parameter names (`LEFT` and `RIGHT`) so they can be identified by the assembler, but the character is not copied to the expanded line.

To include a vertical bar in the expanded line, put two consecutive vertical bars (`||`) into the source line. In general, the expanded line will contain one fewer vertical bars than appear in a sequence of vertical bars i.e., three vertical bars (`|||`) get expanded as two (`||`), and one vertical bar (`|`) gets expanded as none.

8.5 Automatic Label Field Generation

The concatenation operator also has one special usage: if a vertical bar is immediately followed by a pound sign (`|#`), the assembler automatically generates a label field that depends on the number of times the macro has been called. The first time a macro is called, each occurrence of the string "`|#`" is replaced by the string "`A`". On the second call, the string "`B`" is substituted. The assembler provides unique label field strings for the first 65536 calls.

The example in the section on formal parameters illustrated the use of automatic label field generation. The first macro call generated the label `LOOP_A`, and subsequent calls will generate `LOOP_B`, `LOOP_C`, etc.

The user must ensure that the generated labels do not cause multiply-defined errors. If two macros, for example, have the label `LOOP|#` in their models, then both macros will generate label `LOOPA` in the first expansion. Also, if a macro uses the label `LOOP|#`, then the user should not define any labels that concatenate `LOOP` with any combination of four letters in the range A through P.

8.6 MEND

The `MEND` pseudo-op, which marks the end of a macro definition, is explained in Chapter 5. Note that the `MEND` statement must not be labeled.

8.7 MEXIT

The `MEXIT` pseudo-op, which signals the end of macro expansion but not the end of the macro definition, is explained in Chapter 5. `MEXIT` is used in conjunction with conditional assembly (see Chapter 7). The user must exercise caution, however, to ensure that the `MEXIT` statement does not defeat the conditional assembly. In general, `MEXIT` avoids making the assembler expand lines that will be skipped anyway.

The following example illustrates the use of conditional assembly and `MEXIT`. Suppose that we commonly test a loop condition at the bottom and branch conditionally to continue the loop. The range of a branch instruction is short, only `$80` back or `$7F` forward. Sometimes we can get away with coding "`BNE LOOP`", but sometimes the top of the loop is out of range. The following macro handles that problem.

```
MACRO                    Automatically generate a short or long
LOOPNE                 conditional branch to the top of a loop.

GOIF    *+2-\1-          If the destination is out of range of a
        .GT.$80, $1    branch, then go to generate a long jump.
```

```
        BNE     \1                Generate the short branch-not-equal and
        MEXIT                   quit macro expansion.

$1
        BEQ     LNE|#             Since the destination is out of range,
        JMP     \1              branch around a long jump if the contrary
LNE|#                           condition exists.  We would like to do a
                                branch not equal, but the top of the loop
                                is too far away.  Instead, we branch equal
                                around a long jump.

        MEND
```

Notice that there are two possible ways macro `LOOPNE` can be expanded. If the destination address is within byte range, the macro call "`LOOPNE TOP`" will generate a short `BNE` instruction, then the assembler will process the `MEXIT` instruction and stop macro expansion. However, if the destination is too far away, the assembler will be in conditional skip mode when the `MEXIT` instruction is expanded. In skip mode the assembler is looking for the target label (`$1`, in this case) that ends the skip. Thus, the assembler never decodes the `MEXIT` instruction and continues to expand the rest of the lines.

8.8 Macro Call

As the preceding examples illustrate, a macro is called simply by putting the macro name into the opcode mnemonic field of an instruction. If the macro uses parameters, then their substitution strings may be specified in the call. Formal-parameter macro calls are of the form:

```
        <macro_name>    [<param_name>=<string>][,...]
```

where each formal parameter is named and a substitution string is directly assigned. For example,

```
        CLRBLK  FROM = TAB1, -
                TO   = TAB1+TABSIZ-1
```

Since formal parameters are named, they may be assigned in any order. Furthermore, since formal parameters may be assigned a default substitution string in the macro definition, parameters that are not listed in a macro call will be assigned their associated default strings. If a parameter that has no specified default string is not assigned in a macro call, the assembler will assign the null string to that parameter. Such an omission is flagged as a

missing parameter error (see Chapter 10) unless the user has suppressed MP error messages (see Chapters 5 and 6).

A call to a macro that was modeled with formal parameters can also be invoked with the arguments arranged as positional parameters. In other words, if the user is cautious to match the order and number of parameters that were defined in the macro model with the macro call argument strings, then the sometimes bothersome requirement of typing the formal parameter name and its accompanying equals sign can be omitted. The user should be aware that the style of the first argument in the macro call determines the style for the rest of the string substitutions. The assembler does not support a mixed mode of argument specification with respect to formal parameters.

A macro call for a positional-parameter macro is of the form

```
<macro_name>    [<string>][,<string>]...
```

where each string is assigned to the positional parameter at the associated index. The first string is assigned to positional parameter \1, the second string is assigned to positional parameter \2, and so forth. The assembler will assign the null string to any parameters for which the macro call did not specify a substitution string. In that case, the assembler will report a missing parameter error (see Chapter 10) unless the user has suppressed MP error messages (see Chapters 5 and 6).

Sometimes it is appropriate to assign the null string for a parameter. The syntax <> specifies the null string. The user may also use a comma to delimit the end of a null string. For example, the following methods assign three null strings for the given positional-parameter macro.

```
TEST    <>, <>, <>

TEST    ,,<>
```

The following calls produce the same result as the two calls listed above, but the first one generates a too-many-parameters error message and the second generates a missing-parameter error message. Both of these error messages may be suppressed (see Chapters 5 and 6).

```
TEST    ,,,,            Extra comma delimits last null string.

TEST                    Unspecified strings are assumed to be null.
```

String syntax is discussed in a previous section of this chapter.

8.9 Nested Macro Calls

Macros can call other macros. If the assembler is expanding a macro and an expanded line contains a macro call for another macro, the assembler will expand the second macro and resume expansion of the first macro when the second macro expansion is complete. Calling another macro from within a macro is called nesting. Macros may be nested to any level, but macro recursion is not supported at any level. A macro must not call itself, and a macro must not call another macro that, directly or indirectly, calls the macro being defined. If macro recursion is attempted, the assembler reports an error and refuses to expand the illegal macro call.

8.10 Hints for Using Macros

The user must exercise caution in the use of local labels in macros. Notice that none of the previous examples in this chapter uses local labels to avoid giving the impression that it is a good idea. The main problem with using local labels within a macro definition is that it is easy to forget that the label is there. For example, the user might refer to `1B`, meaning the local label that is clearly visible in the source code, and the assembler would reference the closer `1B` if one were inside a macro. For example, consider the following code excerpt:

```
1H      .
        .
        TEST
        BNE     1B
```

If `TEST` contains local label `1H`, the branch following `TEST` will branch into the macro, which is probably not what the programmer intended.

The following example illustrates another problem with using local labels with a macro. Macro `LOOPNE` was previously defined in the section describing the `MEXIT` assembler directive.

```
1H      .
        .
1H      LOOPNE  1B
```

Obviously, the programmer wants a conditional branch to the top of the loop, at the first `1H` local label. We tend to think of macros as occupying a single line when we write the call, but the expanded macro resides on more than one line. Inside `LOOPNE` is a branch to `\1`. Guess where that is. Not where the programmer expected: the `1B` substituted for `\1` points to the macro call itself. This code can hang the program into an infinite loop that never gets out of the expanded macro.

Macros can be used for several purposes. Obviously, anytime a sequence of instructions must be repeated frequently, a macro can save time. However, before defining a macro, the

programmer should consider the functionality of that sequence of instructions. Does the proposed macro perform a function? Or does the group of instructions just happen to fall together often? Would a macro clarify the flow of a program by raising the level of the code? Or would it obscure the meaning of the program by hiding part of the processing?

One common use for macros is to implement opcodes that would be nice to have. For example, the M6800 instruction set is missing many instructions that the M6809 provides. One could define a library of macros that would simulate the 6809 instruction set (or a subset thereof) on the 6800.

Another way of using macros is to streamline subroutine calls. For example, if a subroutine has the following calling sequence,

```
JSR     SUBR
FCB     ARG1
FDB     ARG2
FCB     ARG3
```

the macro

```
MACRO
SUB
JSR     SUBR
FCB     \1
FDB     \2
FCB     \3
MEND
```

allows calls of the form

```
SUB     ARG1, ARG2, ARG3
```

In addition to providing a high-level call statement, such an approach simplifies later changes to the calling sequence. For example, say we want to change the order of the arguments, or we want to push the arguments onto the stack before jumping to the subroutine. We can change the macro definition, and all of the subroutine calls will be revised automatically.

Chapter 9

Relocatable Assembly

9.1 Preliminary Discussion

Before outlining the specifics of relocatable assembly as it has been implemented within the M6800 assembler, it may prove useful to briefly circumscribe the scenario that operationally defines relocation. The digression concerns the process that begins with some high-level language constructs and ends with that unique sequence of binary information that, in the von Neumann sense, implements the algorithm on a specific machine. The user probably knows this process couched in terms of the components to a commonly observed vicious cycle of program development: edit, compile, link, run, edit, compile, link, run, ...

In the specific case that concerns the assembly-language programmer, the high-level language constructs are the opcode mnemonics and pseudo-ops that comprise the assembly language itself. It is the task of the assembler to take as input these constructs and based upon some well-defined syntactic and semantic rules refine the high-level meaning into a sequence of numeric data, known as object code, that will be eventually executed in a specific microprocessor. It is in the realm of the word "eventually" in the last sentence that the concept of relocation is important.

Consider first the simple situation of a dedicated microcomputer without any special memory-management techniques. One hits the reset button, and hardware functions clear the registers and perhaps load the program counter with the address of some simple bootstrap ROM-based routine that first loads, then transfers control to a rudimentary operating system that resides in some basal area of the address space. Say, just to continue a hypothetical situation, that this operating system can accept through a serial port, a sequence of bytes of numeric data that it can store into memory at arbitrary locations as directed by specifically encoded information bytes at the beginning of each of the transmitted sequences. Finally, when the transfer is complete, a user should be able to pass control to some starting address that represents the first instruction in the downloaded program.

By now, the user may have recognized a similarity between this example and the conventional means of using the M6800 assembler in its default absolute object-code generation

mode in conjunction with the monitor utility MUDBUG. The key point to realize about absolute object code generation is that all address references must be completely defined at the end of the assembly process since the downloaded object code contains quite definite information for the monitor with respect to the microcomputer's address space.

Consider next the problems facing a group of individual programmers each developing a separate module of assembly code for a large system. Quite clearly, if each programmer hard wired the data structures and code into specific locations of the memory map, then when the pieces were all put together collisions would almost surely occur. A similar situation exists when developing libraries of routines that a user would like to be able to link in with some main program. To accommodate this degree of freedom in the placement of code in the address space requires another step in the overall assembly process. The assembler, in relocatable mode, defines object code addresses up to additive constant of their ultimate destinations. It then becomes the job of a linker to take several modules of code and resolve ambiguities by assigning a specific value to the base address of the abstract address sections that all relocatable addresses are related to by assembler defined offsets.

One such linker is the LK6800, and the exact nuances of relocation as implemented on the M6800 assembler are defined by the requirements and features of this specific software product. These features will be more completely described in this chapter. The key to remember, however, about relocatable assembly is that addresses will be exactly defined at linkage time, though they must be defined to within an arbitrary additive constant at assembly time.

Some users may wonder where the concept of position independent code falls within this discussion, since it often accompanies the relocation concept in conversation. If the simple microprocessor described above is expanded into a multi-user environment, perhaps with virtual memory, then a case could be made for requiring code that could be executed from any position in the address space. In particular, for code to be sharable, it must be position independent. The key here is that all address references can only be made with respect to some value, usually contained in the program counter or perhaps the stack pointer, that is finally defined at execution time. By keeping all address references ambiguous to this degree allows the loader (that piece of software that deposits a program in memory and transfers control to it, such as the VMS `RUN` command) complete freedom in assigning the code to some available block of memory.

9.2 Invoking Relocatable Assembly

As with most features in the M6800 assembler, the option to select relocatable object code generation can either be done at run time, or with an assembly-time option included in the source code. Of course, if the user does not specify a file or device for the object code, none will be generated; though errors specific to relocatable assembly will still be reported should they occur.

At run time, relocatable assembly may be selected by specifying the `RELOCATABLE` option. The default option `ABSOLUTE` for absolute code generation is otherwise in effect. Refer

to Chapter 6 for more information about using the `RELOCATABLE` and `ABSOLUTE` run-time options.

At assembly time, relocatable assembly may be specified with the "`OPT REL`" pseudo op. Its absolute mode counterpart is "`OPT ABS`". Chapter 5 describes the use of assembly-time options in detail. Note that assembly-time source options have precedence over the run-time variants. It is also important to note that after the first significant statement (i.e., one that defines a label or generates object code) is processed, a request to change object generation mode is flagged as an `AR` error and the current mode is left unchanged.

9.3 Features of M6800 Relocatable Assembly

Relocation is implemented by defining a number of abstract partitions of the microprocessor address space called sections. Each type of section is then restricted to contain code or data of specific attributes so that a program like the LK6800 linkage editor can successfully incorporate the relocatable objects of several modules into one coherent package. At a general level of understanding, relocatable address and expression references are defined to within an additive constant (i.e., the yet to be determined value of the base address of the particular section) and the assembler must leave markers or flags within the object code records to inform the linker where adjustment must be made. A more precise specification of the communication between the LK6800 linker and the M6800 assembler can be found in the documentation for the LK6800 linkage editor, since, for the most part, these specifications remain transparent to the user.

For now, it is enough to describe the five specific types of program sections and their attributes that the M6800 assembler supports in relocation mode. Each of the sections have a corresponding pseudo op that causes the assembler to update the location counter variable that it internally maintains to match the current state of the location counter uniquely defined within each section.

`ASCT`, or absolute section, is a non-relocatable section. There may be a number of absolute sections in a user's program. These sections are used to allocate or initialize memory locations that are assigned by the programmer rather than by the LK6800 linkage editor. `ASCT` can be used to define the locations of PIA's or ACIA's, for example. Since the `ORG` pseudo op is only defined as an absolute construct, its appearance in a source line has the same effect as switching to an absolute section with an `ASCT` statement.

`BSCT`, or base section, is a relocatable section. There is only one base section. The linkage editor assigns portions of the base section to each module that requires space in `BSCT`. The base section is generally used for variables that are to be accessed using the direct addressing mode. `BSCT` is restricted to memory locations `$00`–`$FF`, inclusive. If the location counter for this section ever exceeds this one page limit, the assembler generates a `LO` error.

`CSCT`, or blank common section, is a relocatable section. There is only one blank common section. `CSCT` is similar to blank common used in FORTRAN. The blank common section

cannot be initialized, hence the only legitimate statements in this section are those that request the allocation of variable storage (such as RMB or BSS). Violations of this convention are flagged by a CI error. Regardless of the names that various modules assign to their allocations of the CSCT, the first byte (and hence all subsequent bytes) of each CSCT address the same byte of the physical address space.

DSCT, or data section, is a relocatable section. There is only one data section. The linkage editor assigns portions of this section to each program that requires space in DSCT. DSCT is generally used to contain variables which are in RAM and are to be accessed using the extended addressing mode.

PSCT, or program section, is a relocatable section. There is only one program section. PSCT is similar to DSCT. However, it is generally used to contain program instructions. The use of DSCT and PSCT facilitates creation of programs that reside in ROM but access variables in RAM.

The use of these section swapping pseudo ops is only supported in relocatable mode, and the assembler displays RE errors when it encounters them in absolute mode.

In addition to the five section types defined above, relocatable assembly permits the use of two label definition constructs that enable location and offset information to be passed between separately assembled modules.

An EXPORT, or equivalently XDEF, instruction is used to declare an "externally defined symbol", a symbol that may be referenced by any separately assembled relocatable program. The general form for the EXPORT instruction is:

```
EXPORT  SYMBOL1, -
        SYMBOL2, -
        SYMBOL3
```

The EXPORT statement must not be labeled and the operand field must contain at least one symbol. The user may specify several externally defined symbols in one EXPORT instruction by typing a comma immediately following each symbol; the assembler also accepts blank spaces and/or tabs between the delimiter comma and the first character of next symbol.

Each symbol must be defined elsewhere in the current program. If the symbols contained in the EXPORT instruction are not defined in the program, an IS, invalid symbol, error will be reported.

The IMPORT, or equivalently XREF, instruction is used to declare "externally referenced symbols", symbols that may be referenced in the current source program, but that are defined (via an EXPORT instruction) in some other program. The general form for the IMPORT instruction is:

```
IMPORT  SYMBOL1, -
        SYMBOL2, -
        SYMBOL3
```

The IMPORT statement must not be labeled. Each symbol in the operand field must be declared by an EXPORT pseudo instruction in the program in which it was defined.

The user can specify several externally referenced symbols by typing a delimiter comma following each symbol, and she or he can also type any number of blank spaces between the delimiter comma and the first character of next symbol. As a limitation of the LK6800 linker, only a maximum of 251 symbols can be imported, and the assembler will generate an IML error if this limit is exceeded.

As with the five section swapping pseudo instructions, EXPORT and IMPORT (or XDEF and XREF) are only valid when the assembler is in the relocatable-assembly mode, otherwise, an RE error is reported.

9.4 Implications of Relocatable Assembly

To the user accustomed to the absolute assembly mode, the profusion of program sections and symbol swapping might appear overwhelming. However, in practice, good programming techniques dictate just the type of constant, variable and program statement segmentations that the LK6800 linkage editor facilitates.

After one has accepted the structure imposed by the five types of sections, a possible next step is to chain the context of modules together by passing symbols with IMPORT and EXPORT statements. To aid in keeping the origin and use of these symbols separate from the standard use symbols, symbol table references are specially marked with an I or an E to stand for IMPORT and EXPORT respectively. The structure of the main source listing is also modified by the assembler in relocatable mode to account for the different possible program counters and label section attributes. See the sample listing in the next section for specific details.

The final threshold to attain before mastering relocatable assembly is probably the use in expressions of a mixture of external symbols and so-called relocatable labels defined in the various types of program sections that support relocation. Indeed, in the relocatable mode, the expression evaluating routines in the assembler need to perform a more cautious check to ensure, for example, that 8 bit fields do not contain relocatable expressions which at link time could possibly exceed the field allocated to them (this situation is flagged as an RV error).

More precisely, the following rules must be met in order for the expression to be valid.

- Relocatable symbols or expressions cannot be multiplied, divided, or operated on with the logical operators. This is because the linker (LK6800) can only adjust values by adding or subtracting the load address of the program section (i.e., addresses are defined to within an arbitrary additive constant). Improper operators will cause the assembler to generate an UO error.
- A relocation count is maintained for each program section represented within an expression. Adding a relocatable symbol causes the relocation count to be incremented;

subtracting a relocatable symbol decrements the relocation count. After an expression has been evaluated, the following criteria must be met:

  - All section counts except for one must be zero. This is because the linker can only adjust the object value by one load address. All but one section's load addresses must cancel.
  - The exceptional section must have a count of either zero or one or minus one. This is because the linker can only add or subtract the load address to adjust the output value.
  - When an expression is used within the one-byte immediate addressing mode, the indexed addressing mode, or with the `FCB` directive, all section counts except the `BSCT` count must be zero. This is necessary to be able to guarantee that the adjusted value will fit in one byte. While absolute section (`ASCT`) symbols can be evaluated fully at assembly time, relocatable symbols cannot be fully evaluated since the offset applied at load time can drastically change the symbol's value. However, since base section (`BSCT`) variables cannot be relocated past memory page zero (`$00` to `$FF`), we can allow a `BSCT` variable in a one-byte field.

- Any number of relocatable symbols (including the asterisk denoted current location counter value, and symbols defined in `BSCT`, `CSCT`, `DSCT`, and `PSCT`) are accepted by the expression evaluator but only six external references in an expression are accepted. This is due to a restriction of the linker and violation of this limit will be marked with the `TR` assembler error. Notice that while several relocatable symbols can (and must) cancel by alternate addition and subtraction of their load offsets; the same does not apply to externally referenced symbols.

A final notion related to the use of imported symbols or relocatable labels concerns what is effectively a forward reference. The assembler must evaluate an expression in a statement such as `RMB` or `ORG` or any of the conditional-assembly directives during pass one of the assembly. An imported label or a relocatable label is not fully defined until link time, so an imported label or a relocatable label is not allowed in an expression that the assembler must evaluate during pass one of the assembly. Using an imported label or a relocatable label in an expression that the assembler must evaluate during pass one causes the assembler to report an `IMP` or `RLV` error, respectively.

While the statement of the above rules is intended to convey the rigor of the expression evaluator with respect to relocatable expressions, it perhaps is not so very clear to the user what is implied in the rules. The following listing represents an example assembly file with nothing but symbol definitions and expressions. It comes complete with both errors and examples of correct usage. The examples are far from being exhaustive so the best policy, as usual, is to actually try a questionable statement with the assembler itself.

The file `EXPRESS.ASM` begins here.

```
        OPT     REL             Select relocatable assembly so that the
                                directives are permitted.

        ASCT                    Start an absolute section.  This effect
a1      EQU     10              can also be achieved with an ORG statement.
a2      EQU     20

        CSCT                    Start a common section.  Only allocation
c1      RMB     1               of variables is allowed in a CSCT so no
c2      RMB     1               initializations can be made.

        DSCT                    Start a relocatable data section.
d1      RMB     1
d2      EQU     d1

        IMPORT  im1,im2         Import values from an external module.

        PSCT                    Begin a program section where various
                                expressions will be tried.

onea    EQU     d1-d2           These two expressions have relocation
oneb    EQU     c1-c2           counts of zero since two values with the
                                same relocatable base are subtracted from
                                each other.

onec    EQU     d1-c2           However, this expression is invalid and
                                produces an RL error since two relocation
                                counts are nonzero.

oned    RMB     d1              Trying to use a relocatable value to
                                allocate space produces an RLV error since
                                this information must be known during pass
                                one of the assembly.

onee    EQU     im1             While using a relocatable expression in
                                an EQU directive is permitted, an imported
                                label acts as a forward reference and will
                                cause the assembler to generate an IMP
                                error.

two     LDX     =1*d1           Even though 1*d1 is only d1, this line
                                has an UO error since only addition and
                                subtraction are supported by the linker.
```

```
threea  LDAA    =d1-im1-im2      This expression is well formed with
                                 respect to relocation count, but an 8-bit
                                 field is implied and relocatable values
                                 are therefore not allowed.  An RV error
                                 is reported.

threeb  LDX     =d1-im1-im2      The same expression is fine when it is
                                 slated for a 16-bit field.

threec  LDX     =*-d1-im1-im2    An invalid relocation count error, RL,
                                 occurs here since both the d1 and the *,
                                 which associates with PSCT, have nonzero
                                 relocation counts.

        DSCT

four    EQU     *-d2             If the program section is changed to
                                 DSCT, then the *, location counter symbol,
                                 also changes meaning. Here *-d2 gives a
                                 zero relocation count.

five    FDB     a1+c1-c2-*+im1   Absolute, imported and relocatable
                                 symbols can appear together as long as the
                                 above rules are observed.
```

9.5 Example Usage of Relocatable Assembly

In this section a simple and contrived example of relocatable assembly is presented which demonstrates the major features discussed above. Specifically, two modules, `MAIN.ASM` and `INTRPT.ASM`, were separately relocatably assembled, and then linked with the LK6800 editor to produce an S Record format object file. The source files, an example relocatable assembly listing, and a transcript of the LK6800 linking session are given below. Next, the two modules were combined into one program that was rewritten for the absolute assembly code generation mode. The source file for this combined program is also given. Finally, both of the resultant object files are listed to confirm the comparability of the two processes.

The source file `MAIN.ASM` begins here.

```
        TITLE   MAIN.ASM
        BTEXT

    This is a contrived, but hopefully informative, example of relocatable
assembly techniques.  The program consists  of two modules.  One, contained
in this file, is a main routine which obtains  interactive  input  from the
```

user regarding special interrupt handler routines. The second file, INTRPT.ASM, contains a series of these interrupt handlers.

```
        ETEXT

        OPT     REL             Since relocation will always be needed
                                with this source code, the OPT REL
                                selection is preferred over the front-end
                                RELocatable mode.

        IMPORT  NMI1, NMI2      Bring in the labels that mark the
                                starting points of the interrupt handlers.
                                This is global information at link time.

        ASCT

NMIVCT  EQU     $FFFC           The 6800 microprocessor hardware vectors
                                the NMI through this location. It is an
                                absolute location in the address space.

        CSCT

PARAM   RMB     1               The main routine monitors this variable
                                which is set by an interrupt handlers.
                                This is global information at run time.

        DSCT

PROMPT  FCC     "OPTIONS?"      The data section is used for variables
                                and constants. Here the input prompt is
                                created with the FCC directive.

        PSCT

MAIN    LDAA    =0              The main routine begins here and notice
                                that no ORG statements have been used to
                                "lift" the code above the MUDBUG utility
                                in low memory. This feature will be taken
                                care of by the linker. The LDAA
                                instruction is representative of
                                initialization. In each of these blocks,
                                only one or two statements are given so
                                that the final object file will not be
                                too unwieldy.
```

```
        LDX     =PROMPT         This represents the first step involved
                                in displaying the input prompt at the
                                terminal. Actually, this code could well
                                be in a library of i/o routines in another
                                module.

        CMPA    =1              This comparison shows the last step in
        BNE     1F              a routine which obtained input from the
                                user and left it, converted to numeric
                                data, in the AR.

1H      LDX     =NMI2           Use the relocatable, imported value of
        STX     NMIVCT          the second NMI interrupt handler as the
                                one selected by the interactive input.
                                Note that an absolute address is being
                                referenced here.

WAIT    CMPB    PARAM           Wait in this sense loop until the
        BNE     WAIT            interrupt handler has stored the quit
                                signal in PARAM.

        END
```

The source file `INTRPT.ASM` begins here.

```
        TITLE   INTRPT.ASM
        BTEXT

    This is the companion module to MAIN.ASM. It contains a series of
possible NMI interrupt handlers that the user will select interactively
from within the main routine.
        ETEXT

        OPT     REL

        EXPORT  NMI1, NMI2      These two labels, which are the starting
                                addresses of the interrupt handlers, will
                                be passed to MAIN.ASM via the EXPORT to
                                IMPORT connection.

        BSCT

SHORT   RMB     1               Since interrupt handlers need to be very
```

```
AND     RMB     2               efficient, it may well be that the direct
FAST    RMB     1               addressing mode will be profitably used.
                                Perhaps one of the routines is very memory
                                efficient, and the other is execution time
                                efficient.

        CSCT

GLOBAL  RMB     1                 Even though MAIN.ASM calls this variable
                                PARAM, the accesses to GLOBAL in this
                                module get mapped to the same physical
                                memory address.  This method of global
                                communication is similar to FORTRAN common
                                equivalencies.

        DSCT

TRANS1  FCC     "A"               The data section here contains the
TRANS2  FCC     "B"             beginnings of two translation tables that
                                the two interrupt handlers might use to
                                quickly convert data they obtain from the
                                stack.

        PSCT

NMI1    LDAA    FAST              NMI1 is the fast one. Note that register
                                and stack usage is not explicitly shown
                                here as for any real interrupt handler.

        STAB    GLOBAL            Before returning, NMI1 will store some
                                value in the communications channel between
                                itself and MAIN.

        RTI                       Return from Interrupt.


NMI2    LDAA    SHORT             NMI2 is the short one. Again register
                                and stack usage are not explicitly shown.

        STAB    GLOBAL            Store some value in the communications
                                channel before returning.

        RTI                       Return from Interrupt.
```

```
        END                          End of subordinate module, INTRPT.ASM
```

The file `MAIN.LST` begins here

```
1       25 Oct 1985  19:08   M6800ASM-1.1   MAIN.ASM

    3.
    4.                       This is a contrived, but hopefully informa
    5.                   assembly techniques.  The program consists of t
    6.                   in this file, is a main routine which obtains i
    7.                   user regarding special interrupt handler routin
    8.                   INTRPT.ASM, contains a series of these interrup
   10.                            OPT     REL               Since reloc
   11.                                                     with this sou
   12.                                                     is preferred
   13.
   14.                            IMPORT  NMI1, NMI2        Bring in th
   15.                                                     points of the
   16.                                                     global inform
   17.
   18.    0000                    ASCT
   19.    FFFC             NMIVCT EQU     $FFFC             The 6800 mi
   20.                                                     the NMI throu
   21.                                                     absolute loca
   22.
   23.  C 0000                    CSCT
   24.  C 0000  0001       PARAM  RMB     1                 The main ro
   25.                                                     which is set
   26.                                                     is global inf
   27.
   28.  D 0000                    DSCT
   29.  D 0000  4F         PROMPT FCC     "OPTIONS?"        The data se
        D 0001  50
        D 0002  54
        D 0003  49
        D 0004  4F
        D 0005  4E
        D 0006  53
        D 0007  3F
   30.                                                     constants.  H
   31.                                                     with the FCC
   32.
   33.  P 0000                    PSCT
```

```
   34.  P 0000  86 00    MAIN    LDAA    =0              The main ro
   35.                                                 no ORG statem
   36.                                                 code above th
   37.                                                 This feature
   38.                                                 linker.  The
   39.                                                 of initializa
   40.                                                 only one or t
   41.                                                 the final obj
   42.
   43.  P 0002  CE 0000          LDX     =PROMPT         This repres
   44.                                                 displaying th
   45.                                                 Actually, thi
   46.                                                 of i/o routin
   47.
   48.  P 0005  81 01            CMPA    =1              This compar

2       25 Oct 1985  19:08   M6800ASM-1.1   MAIN.ASM

   49.  P 0007  26 00            BNE     1F              a routine whi
   50.                                                 and left it,
   51.                                                 the AR.
   52.
   53.  P 0009  CE 0000  1H      LDX     =NMI2           Use the rel
   54.  P 000C  FF FFFC          STX     NMIVCT          second NMI in
   55.                                                 selected by t
   56.                                                 an absolute a
   57.
   58.  P 000F  F1 0000  WAIT    CMPB    PARAM           Wait in thi
   59.  P 0012  26 FB            BNE     WAIT            handler has s
   60.
   61.                          END

No errors in pass 2.
7 non-local labels in the symbol table.
1 local label in the symbol table.

3       25 Oct 1985  19:08   M6800ASM-1.1   MAIN.ASM

Concordance   (M = Multiply-Defined Label; U = Unused Label; ----- = U
              (I = Imported; E = Exported)
              (B = BSCT; C = CSCT; D = DSCT; P = PSCT; Blank = Absolut

 Line   Value   Symbol  Reference List
```

```
   34  U 0000 P MAIN
   19    FFFC   NMIVCT     54
   14 IU 0000   NMI1
   14 I  0000   NMI2       53
   24    0000 C PARAM      58
   29    0000 D PROMPT     43
   58    000F P WAIT       59

4       25 Oct 1985  19:08   M6800ASM-1.1   MAIN.ASM

Unused Labels (M = Multiply-Defined Label)
              (I = Imported; E = Exported)
              (B = BSCT; C = CSCT; D = DSCT; P = PSCT; Blank = Absolut

 Line  Value   Symbol

   34   0000 P MAIN
   14 I 0000   NMI1
```

The monitor session with LK6800 begins here. Prompts from the linker are given in `CAPITAL` letters, and the user's responses are given in lower case letters. The notation `<cr>` represents a carriage return.

```
$ LK68 <cr>
SOURCE FILES (OBJ)>  main,intrpt <cr>
OUTPUT FILE (OBJ,MAP)>  final <cr>
START OF PSCT (2000)>  <cr>
START OF CSCT>  3000 <cr>
No error in linkage.
$
```

Some notes on this transcript are appropriate. First of all, LK68 is a command to run the LK6800 linkage editor specifically set up to run on the VAX/VMS at the Arizona State University ECS installation. The actual command name of the linker is site specific.

The source files in this example are the two relocatable object files produced by assembling `MAIN.ASM` and `INTRPT.ASM` with the M6800 assembler in relocatable mode. The output file, `FINAL.OBJ`, will then contain the S Record equivalent of the joined relocatable files.

The linker defaults the `PSCT` base address to `$2000`, though any other value could be chosen. For the `CSCT`, an input value is required and it was arbitrarily chosen as `$3000` for this example. The `DSCT` value is hard coded to `$0100` by the LK6800 linker.

The "No error in linkage." message reports a successful linking.

The file `ABMAIN.ASM` begins here.

```
        BTEXT
```

This is a comment stripped, combined version of the MAIN and INTRPT modules using absolute object code generation. Hopefully, the resulting object files will be identical. It may be useful to point out that the LK6800 linker will default the data sections to $100 and the PSCT to $2000 and therefore these specific values are introduced with ORG statements. The value for the CSCT was selected as $3000 at LK6800 run time as described above.

```
        ETEXT

NMIVCT  EQU     $FFFC

        ORG     $100
PROMPT  FCC     "OPTIONS?"
TRANS1  FCC     "A"
TRANS2  FCC     "B"

        ORG     $0000
SHORT   RMB     1
AND     RMB     2
FAST    RMB     1

        ORG     $3000
PARAM   RMB     1
GLOBAL  EQU     PARAM

        ORG     $2000
MAIN    LDAA    =0
        LDX     =PROMPT
        CMPA    =1
        BNE     1F
1H      LDX     =NMI2
        STX     NMIVCT
WAIT    CMPB    PARAM
        BNE     WAIT

NMI1    LDAA    FAST
        STAB    GLOBAL
        RTI

NMI2    LDAA    SHORT
        STAB    GLOBAL
```

```
        RTI

        END
```

Object file from linkage of `MAIN` and `INTRPT` to form `FINAL.OBJ`

```
S10B01004F5054494F4E533F88
S11320008600CE010081012600CE201AFFFFFCF1DC
S1072010300026FB77
S105010841426E
S10F20149603F730003B9600F730003BC9
S9030000FC
```

Object file from `ABMAIN.ASM`

```
S10D01004F5054494F4E533F414203
S11320008600CE010081012600CE201AFFFFFCF1DC
S1132010300026FB9603F730003B9600F730003B78
S9030000FC
```

The claim is made here that both of these object record files are, in fact, identical with respect to the information passed to some utility loader for the M6800 microprocessor. Of course, since the sizes of each record are not the same across the two versions, checksum values will also be different. For the users who are not convinced, the final part of this chapter explains the fields of a typical record with sufficient clarity to enable the doubters to verify the claim for themselves.

9.6 A Few Words About the S Record Format

For the interested reader, a complete description of the S Record format object code is provided. On a casual or first reading, this material can be omitted since usually it is transparent to the user.

Notice, first of all, that all object values are output in ASCII-coded hexadecimal, so object modules can easily be handled by timesharing systems. Object modules can also be listed and manually interpreted rather easily. The format of an object record is best explained by an example. The following ASCII string illustrates a typical absolute object data record with 16 data bytes.

```
S11301008E0132FE0137C603960AA1022705095A59
```

The first two characters are `S1` to indicate that we have an absolute object data record. These two characters are the same for every absolute object data record.

The next two characters (`13`) are interpreted as a hex number to tell how many more pairs of characters there are in the object data record. This particular record contains `$13` (i.e., 19) character pairs beyond the first four characters, so the total record contains $4 + (19 * 2) = 42$ characters.

The next four characters (`0100`) are interpreted as a 16-bit hex number, and this number is used as the memory origin for the data values that follow in the record. In this example, the data values would be loaded into consecutive memory locations of a microcomputer starting with location `$0100`.

The next two characters (`8E`) are interpreted as an 8-bit unsigned hex number, and this value is a data value to be loaded. `$8E` will be loaded into location `$0100`; `$01` will be loaded into location `$0101`; `$32` will be loaded into location `$0102`; . . . ; and `$5A` will be loaded into location `$010F`. There are 16 data bytes in a full object data record, so object files of full object data records can easily be broken down into segments for individual ROMs, PROMs, or EPROMs.

The last two characters of the example object data record (`59`) are interpreted as an ASCII-coded hexadecimal number, and this number is the one's-complement 8-bit checksum for the object data record. `$59` is therefore the 8-bit one's complement of the 8-bit two's-complement arithmetic sum of `$13` + `$01` + `$00` + `$8E` + `$01` + `$32` + . . . + `$5A`. Notice that the checksum includes the record-length characters and the block-origin characters as well as the object data values. Motorola may have used a one's-complement checksum instead of a two's-complement checksum just to make their object format different from Intel's 8080 object format, or they may have used a one's-complement checksum because a two's-complement checksum allows a record of all zeros to pass. In any case, the standard Motorola object format requires a one's-complement checksum.

Chapter 10

Assembly Errors

This chapter is confined to assembly errors, those abnormalities of source line syntax and semantics found in the assembly process itself. As such, no discussion is made of possible errors and the messages that occur during the interactive front-end processing aspect of the assembler's execution. Information about file handling errors at the system level or incorrect run-time option specifications is discussed in Chapter 6 itself. As such, no discussion is made of possible errors and the messages that occur during the interactive front-end processing aspect of the assembler's execution. Information about file handling errors at the system level or incorrect run-time option specifications is discussed in Chapter 6 on running the M6800 assembler.

The M6800 assembler produces an error line for each error it finds up to a limit of five errors per line. If the `NReport` option is specified errors are not reported to the symbolic listing. If the `NError` option is also selected, errors are not reported to the terminal, either. More information on these two run-time options and their interaction can be found in Chapter 6. The user should be careful in the selection of the two error-suppression options since fatal errors may be hidden from the user. The default error reporting is to send the error messages both to the assembly listing (if one has been specified) and to the terminal. If the terminal has been selected as the listing output device, the errors are reported only once to avoid confusion and time consuming duplication.

A message that shows the total number of errors that were recorded during the assembly is output at the end of pass 2. When a complete pass-2 symbolic listing is output, all the error lines in the listing are chained together (via their line numbers) into a linked list of sorts, so the user can easily find all of the errors in a program even if the program is quite long. The user can also use a text editor to search for the `??XX` string that is used to identify each error message. In any case, the user should be able to locate all errors quite easily.

A typical error line consists of two fields. The first, as alluded to above, contains characters of the form, `??XX`, where `XX` represents a unique two or three letter error mnemonic corresponding to each of the possible assembly errors. Besides the convenience of the `??XX` form in conjunction with a search command in a text editor, the appearance of a unique

code within the first few characters of a line ensures that these important messages will not go unheralded even in a severely truncated listing.

The second field of the error line contains a complete English sentence which offers a concise explanation of the assembly error. The message is short enough to occupy only one line of the listing and in fact is kept under 80 characters to avoid the sometimes distracting line wrap around on standard CRT display devices.

All possible assembly error messages are reproduced below in alphabetic order for convenience of the user. Each message is accompanied by an explanation of the error, probable causes of the error, and when appropriate, a cross reference to other chapters of this user's guide where further relevant examples of correct usage appear.

`??AD The operand address exceeds the range allowed for this instruction.`

ADdressing error. A branch-class instruction is coded with a destination address that is outside the range of addresses that can be reached by the instruction.

`??AM The given addressing mode doesn't exist with the given instruction.`

Addressing Mode error. The user has correctly specified the addressing mode, and the user has also correctly specified an instruction. However, the given addressing mode is not available with the given instruction.

`??AR It's too late to specify absolute or relocatable mode.`

Absolute/Relocatable error. An "`OPT ABS`" or "`OPT REL`" statement occurs after the first non-introductory statement, and we can't change modes after we've already started building the symbol table or generating object values.

`??BM A macro name must start with an alphabetic character.`

Bad Macro name. The first non-"white space" character on the line following the `MACRO` pseudo-op is not alphabetic.

`??BR An invalid numeral for a specified base appears in an expression.`

Base Range error. A hextet that appears in a number is too big for the number's base. For example, an "8" or a "9" cannot appear in an octal number, and similarly an "A" or a "B" cannot appear in a decimal number.

`??BT The BTEXT or ETEXT statement is redundant.`

`BTEXT/ETEXT` error. A `BTEXT` or `ETEXT` statement appears redundantly, so a `BTEXT` or `ETEXT` statement is probably missing somewhere.

`??CI Object values are not permitted in CSECT memory.`

`CSCT` Initialization error. Any directive or instruction that generates an object value for the blank common area is illegal. Blank common is shared by all modules as a read/write area of memory, so no module is allowed to contain object values that are assigned to blank common. If two or more modules contained object values for blank common, the object values from the second module would overwrite the object values from the first module. To prevent this possibility, the assembler prohibits any module from generating object values for blank common.

`??CO The first character of this source line is invalid.`

COmment error. The first character in the source line is not valid, so the line is treated as a comment line. Valid first characters are: a tab, blank, a letter (for a symbol), a digit (for a local label), an asterisk (for a comment line), or a dollar sign (for a conditional-assembly label). A carriage return may also be the first character in a line, in which case the line is treated as a blank comment line.

`??DF Divide fault or division by zero occurs in an expression.`

Divide Fault error. Divide fault occurs when a user attempts to divide `$8000` by -1, thus generating a result that doesn't fit correctly into 16 bits. We also report this same error when a user divides by zero in an expression. In either case, the assembler ignores the division and retains the original expression value.

`??DI Direct addressing is not available with the given instruction.`

DIrect addressing error. Direct addressing is requested by the use of an "!" to start the operand field, and direct addressing is not available with the given instruction.

`??DP The operand value exceeds 8 bits.`

Direct-Page register error. The format of the operand expression for a DPR pseudo instruction is incorrect. The proper format is `$00XX`, where `$XX` is the low byte of the operand expression and denotes the value that the user wants for the direct-page register. The high byte of the operand expression must always be zero.

`??EO An expanded macro line contains too many characters.`

macro Expansion Overflow error. The source-line buffer (`SRCLIN`) has overflowed during macro expansion because the expanded line is too long. Perhaps only comments remain on the current line, but we report an error to draw the user's attention to a possible problem.

`??ER A right parenthesis appears with no matching left parenthesis.`

Extra Right parenthesis. A character that functions as a right parenthesis appears in an expression with no matching character that functions as a left parenthesis. The assembler ignores the extra right parenthesis.

`??ES A null string is not a valid operand for this statement.`

Empty String error. An `FCC` statement has no string at all, or it has an empty string such as "" or ''. Since there is no string to process, the assembler processes the statement as if it were an "`EQU *`" instruction.

`??FR A forward reference appears where it is not allowed.`

Forward Reference error. An expression contains a forward reference, but the assembler must evaluate the expression accurately during pass 1. The assembler obviously can't evaluate the expression accurately during pass 1 if the expression contains a forward reference.

`??IA A macro parameter assignment statement is missing the equal sign.`

Invalid macro parameter Assignment error. The next non-"white space" character following a formal parameter name in a macro call statement was not the equal sign as required.

`??IC A continued macro parameter string is followed by a blank line.`

Invalid string Continuation error. A blank line follows a continued macro parameter substitution string. The valid portion of the string will be used.

`??ICH An invalid character appears in an expression.`

Invalid CHaracter. A character that is not valid in an expression appears in an expression. The assembler ignores the invalid character and continues to evaluate the expression as if the invalid character weren't there.

`??II Invalid macro parameter index value.`

Invalid macro-parameter Index error. A backslash is followed by something other than a decimal number in the range from 1 through 255. Positional parameters are referenced as `\n`, where `n` is the position of the parameter in the macro call. This error can occur if an invalid digit string (or no digit at all) follows a backslash in the macro model.

`??IL The label field for this statement must be empty.`

Invalid Label error. A label or a local label appears in the label field of the statement, and the statement is a statement for which a label is not allowed. For example, a label is

not allowed with a `SPACE` statement or an `EJECT` statement. If the invalid label is a local label of the form `jH`, then any local label of the form `jF` (with the same `j` value) in the expression field of the statement has been incorrectly evaluated because of the effect of the invalid label.

`??IM Immediate addressing is not permitted with this instruction.`

IMmediate addressing error. Immediate addressing is specified for an instruction that does not have immediate addressing.

`??IML Too many imported labels.  Only 251 are allowed.`

Importing too Many Labels error. The program contains too many externally referenced symbols. No more than 251 symbols may be specified as external references.

`??IMP An imported label is used where it is not allowed.`

IMPorted label error. An imported label appears in an expression that must be evaluated correctly during pass 1, and the assembler obviously can't evaluate the expression completely when it contains an imported label.

`??IN6 The 6502 doesn't work right when the pointer address ends with $FF.`

The user has specified a 6502 `JMP` instruction with the (`$1234`) indirect addressing mode, and the address of the indirect pointer is of the form `$xxFF`. The 6502, believe it or not, doesn't work right with such an instruction, so we warn the user. The 6502 correctly gets the LSB of the indirect address from location `$xxFF`, but it incorrectly gets the MSB of the indirect address from location `$xx00` instead of getting it from location `$xxFF+1`. The processors in the 65150 group don't have this error.

`??IN The specified file could not be included as source input.`

INclude error. An error has occurred while attempting to access a file as source for an `INCLUDE` statement. Probably, the file name in the operand field is in error, though perhaps a system error occurred on opening or connecting to this file (it would also have been reported in this case). Another possibility is that the file was already opened (i.e., an improper nesting of `INCLUDE` statements) or that no virtual memory was available to accommodate the Include control blocks in the front-end.

`??IO An invalid option or option range has been specified.`

Invalid-Option error. An invalid option mnemonic is specified, or an option value is outside the allowed range for its field.

`??IP A parameter name must begin with an alphabetic character.`

Invalid Parameter name error. A nonalphabetic character occurred where a formal parameter name should begin. The macro call statement is malformed in some way. Perhaps the user attempted to list positional parameter names (of the form `\n`) in a macro call model statement.

`??IS An invalid symbol appears in the symbol list.`

Invalid Symbol error. An invalid symbol appears in a symbol list, and the assembler skips the invalid symbol but continues to process the rest of the symbol list. This error occurs with the `XDEF` and `XREF` statements.

`??IT A tab stop cannot be less than 1 or greater than 80.`

Invalid Tab stop. A tab stop in a column less than 1 or greater than 80 is specified in an "`OPT TABS = n1, n2, ..., nj`" statement.

`??IX An improper base address or index register has been specified.`

IndeXing error. A negative base address is specified with indexed addressing and/or "`X`" is not specified correctly as the index register.

`??LL The local label in the label field is malformed.`

Local-Label definition error. Column 1 contains a decimal digit, thus indicating the presence of a local label in the label field, but the syntax for the local label is incorrect. The correct syntax is `jH` where $0 \leq j \leq 9$.

`??LO The location counter exceeds $FF in the base section.`

Location counter Overflow error. The base section's location counter overflowed its maximum value, `$FF`.

`??LOP An invalid logical operator appears in an expression.`

Logical OPerator error. A logical operator that is not recognized appears in an expression. For example, the user might have typed `.ANC.` instead of typing `.AND.` as intended. The assembler assumes a `.AND.` operator for the invalid logical operator and continues to evaluate the expression.

`??LOT The termination character for a logical operator must be a period.`

Logical Operator Termination error. The termination character for a logical operator

is invalid. For example, a user might have typed `.AND`, instead of typing `.AND.` as desired. The assembler ignores the invalid termination character and assumes a period in its place.

`??LR A local label reference exceeds the local label range.`

Local-label Range error. The instruction refers to a local label that is not defined within `LLRNGE` lines. The user can set `LLRNGE` to any desired value via an `OPT` statement.

`??MC Assembler syntax requires a comma that is missing.`

Missing-Comma error. The assembler's syntax requires a comma, and the comma is missing.

`??MD A label that is defined here is also defined elsewhere.`

Multiply-Defined symbol error. The label that appears in the label field is a multiply-defined symbol. That is, the label that appears in the label field of this statement also appears in the label field of at least one other statement. The user can induce multiply-defined symbol errors in a seemingly correct program by using an `OPT` statement to set `LBLLIM` to a value that is too small; only the first `LBLLIM` characters of each label are used to distinguish labels from each other, and subsequent characters are effectively ignored. Therefore, any labels that are identical in the first `LBLLIM` characters are considered to be the same symbol.

`??ME No END statement has been found to mark the end of the source file.`

Missing-End error. The end of the source file has been encountered, and no `END` statement has been found. Perhaps the user employed a `BTEXT` statement but forgot to include a matching `ETEXT` statement, or perhaps the user employed conditional assembly but did not include the desired target label.

`??MI Immediate addressing is required for the first operand field.`

Missing-Immediate error. Immediate addressing is required for the first operand field of an HD6301 immediate-to-memory instruction, and the user has omitted the immediate-addressing symbol. The assembler automatically uses the correct addressing mode but reports the error to alert the user to the incorrect syntax.

`??ML Missing parameter string delimiter in a macro call.`

Missing deLimiter error. A parameter substitution string that begins with a delimiter is not ended with a delimiter before the end of the line.

`??MM The left and right parentheses aren't the same style.`

MisMatch error. The style of the right parenthesis doesn't match the style of the left parenthesis. For example, a parenthesized subexpression might be enclosed by `(` and `}` or `<` and `]` instead of being enclosed by parentheses of the same style. The user has probably made a typing error.

`??MN A radix symbol has been used but was not followed by any numerals.`

Missing-Number error. A prefix character such as a `$`, `@`, or `%` has been found in an expression to indicate that a number in a particular base should follow, but there is no number. The assembler uses zero as a default value for the missing number.

`??MO The operand field is missing, and it is required.`

Missing Operand error. The operand field for an `FCC`, `ASCII`, `ASCIIZ`, or `ASCIIC` statement is missing entirely.

`??MOP An operator is missing in an expression; multiplication assumed.`

Missing OPerator error. An operator that should appear between two terms in an expression is missing. The assembler assumes a multiplication operator by default and continues evaluating the expression. A parenthesized subexpression is equivalent to a term, so an expression like $2(3+4)$ evaluates to $2*7$ or 14.

`??MP The macro call is missing a parameter string.`

Missing Parameter error. The macro call statement contains fewer positional parameter strings than required, or the call does not specify the string to be substituted for a non-default formal parameter. In either case, we substitute the null string. The user can turn off this error message by specifying the `NOMPERR` option.

`??MQ The statement is missing a literal character argument.`

Missing Quoted character. An apostrophe or a quotation mark has been found in an expression to indicate that the next character is to be taken literally. That is, the ASCII code of the next character is to be used as the value of a term. However, the apostrophe or quotation mark is the last character of the source line, and the next character is missing. The assembler uses a blank space as a default character in place of the missing quoted character.

`??MR Macro Recursion is not allowed at any level.`

Macro Recursion error. During macro expansion, recursion was detected. Either the macro called itself, or a subordinate macro called one of its predecessors.

`??MR6 A required right parenthesis is missing or misplaced.`

The assembler's syntax requires a right parenthesis to close a specification for indirect addressing, and the right parenthesis does not appear where it belongs.

`??MRP A left parenthesis appears with no matching right parenthesis.`

Missing right parenthesis. A left parenthesis appears in an expression with no balancing right parenthesis to close the parenthesized subexpression. The assembler automatically supplies the missing right parenthesis at the end of the expression.

`??MS A label in the operand field is a multiply-defined symbol.`

Multiply-defined Symbol error. A label that appears in the operand field is a multiply-defined symbol. That is, a label that is used in the current line is defined in at least two different places, so the value that is used for the label may not be the value that is desired.

`??MT An expression is missing a term or is otherwise malformed.`

Missing-Term error. An expression ends with an operator that is not followed by a term, or else an expression field is completely empty. Perhaps a blank space was inadvertently typed between an operator and the next term of an expression, or perhaps two consecutive commas were inadvertently typed. Another common cause of this error is the presence of a comma after the last expression field of an instruction that can have multiple expression fields. The comma indicates the presence of another expression field, but the last expression field is missing. The assembler uses zero as a default value for any term that is missing.

`??MX MEND and MEXIT are invalid except in a macro.`

Illegal Macro termination error. A macro end (`MEND`) or macro exit (`MEXIT`) pseudo-op appears outside a macro definition, i.e., in the normal program flow.

`??MY A macro definition is different in pass 2; synchronization error.`

Macro sYnchronization error. On the second pass, we were unable to match the macro definition with a macro node with the same information.

`??NL The nesting level for parentheses is too deep.`

Nesting-Level error. The nesting level for parentheses in an expression is too deep. The assembler ignores the excessive levels of parentheses. The maximum allowable nesting level can be increased to virtually any level by changing the label `MAXNEST`.

`??NS A number exceeds 16 bits when it is converted to binary.`

Number-Size error. A number exceeds 16 bits when it is converted to binary. This error is repeated for each extra digit (or hextet or octet or bit) of the number after the number once exceeds 16 bits.

`??OM Out of memory; ;unable to process macro definition completely.`

Out of Memory error. While processing a macro definition, a request for more symbol-table memory was denied.

`??ON Out of memory; null strings used for specified parameter strings.`

Out of memory — Null strings used error. A request for additional symbol-table memory was denied, so we are unable to store the default parameter substitution strings.

`??OP The string in the opcode field is not a valid opcode mnemonic.`

OPcode error. The string in the opcode field is not recognized as a valid opcode mnemonic.

`??QT A quoted string is improperly terminated.`

Quote Termination error. Something is wrong with the termination of the quoted string in an `FCC` statement. Either the closing string delimiter is missing, or else the closing string delimiter is present but is immediately followed by a nonblank character. An omitted closing string delimiter is obviously an error. A closing string delimiter that is immediately followed by a nonblank character indicates a probable error in which the user accidentally used the delimiter character as a character in the quoted string. In the case of an omitted string delimiter the assembler processes the string to the end of the source line. In the case of a nonblank character immediately following the closing delimiter the assembler processes the string to the closing delimiter.

`??RE Relocation directives are not allowed in absolute mode.`

RElocation directives error. A relocation assembler directive such as `ASCT`, `BSCT`, `CSCT`, `DSCT`, `PSCT`, `IMPORT`, or `EXPORT` appears in a program that is being assembled in the absolute mode. The assembler reports an error and essentially treats the invalid relocation directive as if it were a comment line.

`??RL An expression's relocation factor is invalid.`

ReLocation factor error. The relocation factor for an expression must be in the range from -1 through $+1$, and the relocation factor must be zero for all except one section. We report an error if these restrictions are violated.

`??RLV A relocatable expression value is used where it is not allowed.`

A relocatable expression value appears in a context where it is not allowed. For example, a relocatable expression value is not allowed with `BSS`, `RMB`, `ORG`, `GOIF`, `GOUNLS`, `GOIFZ`, `GOULZ`, or `GOTO` statements, and a relocatable expression value is not allowed as the first expression with the `RCB` and `RDB` statements. The assembler must evaluate these expression values completely and can't defer the final evaluation to the linker.

`??RV A relocatable or imported value is not allowed for an 8-bit field.`

Relocatable Value error. An expression that is used for an 8-bit field contains a symbol that is defined in another module, or the expression has a relocatable program section value that is not `ASCT` or `BSCT`. The assembler doesn't allow such values for 8-bit fields because relocation could cause such a value to exceed 8 bits at link time.

`??SE A synchronization error between passes has occurred for this label.`

Synchronization Error. The value that the assembler computed during pass 2 for the label in the label field doesn't match the value (if any) that was recorded in the symbol table for the label during pass 1. Synchronization errors are usually caused by forward references in expressions that must be evaluated during pass 1. For example, a forward reference in an `EQU` statement or an `ORG` statement or an `RMB` statement would create problems. The root cause of a synchronization error is frequently at some statement before the statement at which the synchronization error occurs. For example, a forward reference in an `ORG` or an `RMB` statement could cause synchronization errors for subsequently defined labels. Synchronization errors can also be caused by errors that are due to forward references in the definitions of labels that are used for controlling conditional-assembly operations. Additionally, synchronization errors arise if symbol-table overflow occurs because label values can't be put into the symbol table after symbol-table overflow has occurred.

`??ST This label is undefined due to symbol-table overflow.`

Symbol-Table overflow error. The amount of memory that the assembler has available for the symbol table is insufficient for assembling the program, and the label in the label field of the current statement cannot be put into the symbol table. This label is therefore not defined, and undefined-symbol errors occur wherever this label is used. The user can use an `OPT` statement to reduce `LBLLIM` to a smaller value, or the user can use fewer labels or shorter labels or more local labels, or the user can deselect the concordance (i.e., cross-reference) option. Another alternative is to break the program into sections that are assembled separately.

`??SZ An expression value is outside the range allowed for its field.`

SiZe error. An expression value is too large to fit into the field in which it is used. The expression value is truncated on the left so it will fit into the desired field.

`??TM A macro call contains too many parameters.`

Too Many parameters error. When retrieving parameter substitution strings for each of the defined parameters, the macro call statement appeared to offer at least one more string. This probably occurred because the user delimited the final parameter with a comma. The user can turn off this error message by specifying the `NOTMERR` as an `OPT` or with `NTMerr` as a run-time option.

`??TO Two consecutive operators appear in an expression.`

Two consecutive operators appear in an expression. The assembler uses the second operator and discards the first operator. A user might have written an expression like $+-3$, for example.

`??TR An expression contains too many imported symbols.`

Too many externally defined symbols Referenced error. Six imported symbols may be referenced in an expression.

`??UL A local label that appears in the operand field is undefined.`

Undefined Local label. A local label that is used in the operand field is not defined anywhere in the program.

`??UO Only addition and subtraction are allowed with relocatable values.`

The assembler allows a user to add and subtract relocatable or imported values, but it doesn't allow any other operations (such as multiplication, division, or logical operations) with relocatable or imported values. The linker can handle only addition and subtraction, so the assembler must not allow any other operations.

`??UP A macro call statement contains an unrecognized parameter name.`

Unrecognized Parameter error. A string that is parsed as a parameter name does not match any defined parameter name. Perhaps the user misspelled a parameter name, or the user attempted to list the substitution string as though the macro uses positional parameters, or the last parameter was delimited by a comma and we treated a comment as a parameter name.

`??US A label that appears in the operand field is undefined.`

Undefined Symbol error. A label that appears in the operand field is undefined. A default value of zero is used.

`??WS A nonblank character appears where white space is expected.`

White-Space error. A field that must be terminated by white space is terminated by a nonblank character. For example, a user might erroneously say "`LDAA =0, X`" where the comma following the immediate value is invalid.

Chapter 11

Summary of Nonstandard Features

The primary differences between this M6800 assembler and the standard Motorola definition of the M6800 assembler are listed below. Some of the points are merely informative, but should be of interest to most users. Please note that any reference to the "Motorola" assembler refers to the M68SAM 6800 assembler that was produced by Motorola, Incorporated. Motorola has now produced other (and better) versions of their 6800 assembler, and deficiencies that exist in the M68SAM Motorola assembler may not exist in newer Motorola assemblers.

- Immediate addressing can be specified by an `=` as well as by the Motorola-defined `#` character.
- A quoted character in an expression can be specified, for example, as `'X` (Motorola's definition), or as `'X'`, `"X`, or `"X"` (new additions).
- The `A` or `B` opcode extension must be adjacent to the opcode when it is used. No intervening spaces are permitted. For example, "`LDA A`" is illegal, but `LDAA` is recognized. As a result of this rule, users may define `A` and `B` as ordinary labels in their programs. With the Motorola version of the assembler, these are special characters and may not be used as labels.
- The Motorola assembler recognizes "`LDAA X`" as being equivalent to a "`LDAA 0, X`" instruction, but this assembler requires the second form. As a result, this assembler permits `X` to be used as a label whereas Motorola does not.
- The assembler doesn't permit the use of suffixes on numbers to specify bases. By eliminating the number suffixes and recognizing prefixes only, the assembler eliminates several exceptions and special cases that the user must consider when using suffixes. The allowed prefixes are `%` for binary numbers, `@` or `O` for octal numbers, the digits 1 through 9 for decimal numbers, and `$` for hexadecimal numbers. We provide the `@` for octal numbers to be consistent with the Motorola definition.

- If the first character in the operand field is an exclamation point (e.g., "`LDAA !SAM`"), this assembler generates an instruction that uses direct addressing. This feature allows the user to dictate direct addressing when desired instead of letting the assembler choose whether to use direct or extended addressing. If an `!` is not included in the instruction, the assembler chooses the addressing mode automatically much as the Motorola assembler does.

- The assembler accepts a string of label characters following a `?` as a term in an expression, and its value is the 16-bit RAD-40 code of the character string. The RAD-40 code value is the same code that the assembler uses internally.

- The assembler treats empty lines as comment lines in the source input. An empty line occurs whenever a carriage return is found with no effective characters preceding it. Otherwise, the assembler ignores empty lines.

- Motorola's definition of the character string with an `FCC` instruction is ambiguous. With their definition, for example, `25,2` could mean "2" followed by 24 blanks, or it could mean "`5,`" with the character 2 as a string delimiter. To eliminate this ambiguity, we do not allow the count-comma-text definition of a string. A string in an `FCC` instruction is therefore introduced by any nonblank character that the assembler accepts for input, and it is terminated by the next occurrence of the same character. The following examples illustrate ways to code `FCC` strings:

```
'THIS IS A SIMPLE STRING'
*ISN'T IT THOUGH?*
XLOOK AT ALL THESE %&$#"!()'& CHARACTERS!X
```

- The assembler requires one or more blank spaces or horizontal-tab characters between any two fields of an assembly-language statement. The assembler has pre-set tabs consistent with the PDP initial tab stops, i.e., 9, 17, ..., 73. The tab stops may be changed by the user with the `OPT TABS` directive (see Chapter 5).

- This assembler does not permit forward references to future symbols in some pseudo-opcode instructions. For example, "`SAM EQU JOE`" is valid only if the label `JOE` has been defined before the `EQU` statement is encountered in the source stream. Refer to Chapter 4 for a discussion of the pseudo-opcodes and which may not contain forward references.

- The front-end processor allows users to select file names and run-time options interactively with the assembler, and the user doesn't need to wait until the end of a run to find out that the options or file names were not acceptable to the assembler.

- The `BLO` (branch lower than) and `BHS` (branch higher or same) opcodes have been added. The `BLO` instruction is equivalent to the `BCS` instruction, and the `BHS` instruction is equivalent to the `BCC` instruction. These two opcodes are provided to make life easier for the user who wants to make comparisons between unsigned values.

- The other primary differences between this assembler and Motorola's definition of the assembly language concern pseudo-opcodes, `OPT` options, and expressions. These items are all thoroughly documented in the previous chapters.